NATIONAL OPEN UNIVERSITY OF NIGERIA

SCHOOL OF SCIENCE AND TECHNOLOGY

COURSE CODE: CIT 703

COURSE TITLE: Information Technology and Software Development

Course Code	
Course Title	Information Technology and Software Development
Course Developer/Writer	Eze, Festus Chux Department of Computer Science Ebonyi State University Abakaliki
Course Editor	
Programme Leader	
Course Coordinator	

Introduction

Information Technology and Software Development is a three credit load course for all the students offering Post Graduate Diploma (PGD) in Computer Science, Information Technology and other allied courses.

Software Development is a major branch in computing and information Technology. A software development professional oversees the processes of software development, the management of software development project, the maintenance of the installed software in an organisation. For sometime the field has been dominated with what is the definitive process of software development. Furthermore there has been the running battle between professionals and managers on who should control a software development project. There is an attempt to classify it as any other project that an organisation handles hence anybody could manage it. Whereas others see it as a highly professional issue that requires high precision in design, management and implementation. However, software development is all involving. It involves the user (client) whose interest is paramount. The developing organisation and her professionals (team)are of great importance. Therefore a successful exercise can only take place when all these variegated interests are harmonised.

What You will Learn in this Course

This course consists of units and a course guide. This guide gives you a brief insight into what the course is all about, the necessary course materials and how you can work with them. In addition it advocates some general guidelines for the amount of time you are likely to spend on each unit of the course in order to complete it successfully.

It gives you guidance in respect of your Tutor Marked Assignment which will be made available in the assignment file. There will be regular tutorial classes that are related to the course. It is advisable for you to attend these tutorial sessions. The course will equip with the wherewithal needed to manage a software development project.

Aim of the Course

The course is designed to provide you with the understanding of managing a software development design and development project. Also it makes you to understand how to take the client into full consideration in the course of the project.

Course Objectives

To achieve the aims set out, the course has a set of objectives. Each unit has specific objectives which are included at the beginning of each unit. You should read these objectives before you study the unit. It is also advisable that you refer to them as you progress in your study of the unit to ascertain your progress. Also you should also look at the objectives after completion of each unit. By so doing, you would have followed the instructions in the unit.

Below are the comprehensive objectives of the course as a whole. By meeting these objectives, you should have achieved the aims of aims of the course as whole. In addition to the aims above, this course sets to achieve some objectives. Thus after going through the course, you should be able to:

Explain the concept of software development life cycle
Identify the different software development models
Identify the software development processes
Explain how to handle software project planning
Explain the software development modelling languages
Explain the concept and application of Data Flow Diagrams
Explain the process of software validation, verification and testing
Explain the issues of state transitions and state chart
Explain the process of requirement engineering, datamodel and dbms

Working through this Course

To complete this course you are required to read each study unit, read the textbooks and read other materials which may be provided by the National Open University of Nigeria.

Each unit contains self-assessment exercises and at certain points in the course you will be required to submit assignments for assessment purposes. At the end of the course there is final examination. Below you will find listed all the components of the course, what you have to do and how you should allocate your time to each unit in order to complete the course on time and successfully.

This course demands that you spend a lot of time to study. My advice is that you optimise the opportunity provided by the tutorial sessions where you have the opportunity of comparing your knowledge with that of your colleagues.

The Course Materials

The man components of the course are:

1. The Course Guide
2. Study Units
3. References/Further Readings
4. Assignments
5. Presentation Schedule

Study Unit

The study units in this course are as follows:

Note each unit consists of one or two weeks' work and includes introduction, objectives, reading materials, exercises, conclusions and summary, Tutor Marked Assignment (TMAs), references and other resources. The unit directs you to work on these exercises related to required reading. In general these exercises test you on the materials thereby assisting you to evaluate your progress and to reinforce your comprehension of the material. Together with the TMA these exercises will help you in achieving the stated learning objectives of the individual units and of the course as a whole.

Presentation Schedule

Your course materials have important dates for early and timely completion and submission of your TMAs and attending tutorials. You should remember that you are required to submit all your assignments by the stipulated time and date. You should guard against falling behind in your work.

Assessment

There are three aspects to the assessment of the course. First is made up of self-assessment exercises, second consists of the TMA and third is the written examination/end of course examination.

You are advised to do the exercises. In tackling the assignments, you are expected to apply information, knowledge and techniques you gathered during the course. The assignments must be submitted to your facilitator for formal assessment in accordance with the deadlines stated in the presentation schedule and the assignment file. The work you submit to your tutor for assessment accounts for 30% of your total course work. At the end of the course you will need to sit for a final examination or end of course examination of about three hour duration. This examination will count for 70% of the total course mark.

Tutor-Marked Assignment

This the continuous assessment component of your course. It accounts for 30% of the total score. You will be given four TMAs (4) to answer. Three of these must be answered before you are allowed to sit for the end of the course examination. The assignment questions for the units in the course are contained in the assignment file. You will be able to complete them through your reading the information contained in the reading materials, references and the study units. You are advised to research deeper into topics so as have a broader view of the discussions.

Endeavour to get the assignments to the facilitator on or before the deadline. If for any reason you cannot complete the work on time , contact your facilitator before the assignment is due to discuss the possibility of an extension. Extension will not be granted after the due date has passed unless on exceptional circumstances.

Final Examination and Grading

The end of course examination for Information Technology and Software Development will be for about 3 hours and it has a value of 70% of the total course work. The examination will reflect the type of self-testing, practice exercise and tutor marked assignment problems you have previously encountered. All these areas of the course will be assessed.

Use the time between finishing the last unit and sitting for the examination to revise the whole course. It might be useful to review your self tests, TMAs and the comments on them before the course examination. The end of course examination covers information from all parts of the course.

Course Marking Sceme

Assignment	Marks
Assignments 1-4	Four assignments,best three marks of the four count at 10% each – 30% of course marks
End of course examination	70% of overall course marks
Total	100% of course materials

Facilitators/Tutors and Tutorials

There are 21 hours of tutorials provided in support of this course. You will be notified of the dates, times and venues of these tutorials as well as the name and phone numbers of the facilitator, as soon as you are allocated to a tutorial group.

Your facilitator will mark and comment on your assignments, keep a close watch on your progress and any difficulties you might face and provide assistance to you during the course. You are expected to mail TMA to your facilitator at least two working days before the schedule date. The TMA s will be marked by your tutor returned back to you as soon as possible.

Do not delay to contact your facilitator by telephone or email if you need assistance.

The following might lead to your needing your facilitator's assistance:

You do not understand any part of the study or assigned reading
You have difficulty with the self test
You have a question or a problem with an assignment or with the grading of an assignment

Endeavour to attend tutorials. It affords you the opportunity of face to face contact with the facilitator and to ask questions which are answered instantly. You also raise problems encountered in the course of study.

Summary
Information Technology and Software Development is a course designed to equip you with what it takes to handle large software project. Further more you learn how to manage software systems in the house. Also you get to understand how you can design your software t o meet the clients needs avoid errors which are very expensive in software development.
Furthermore by the time you are through with this course you can function effectively as a system analyst, software developer, software evaluator etc

I wish you the best and believe you will find the material very interesting.

Contents

1.0 INTRODUCTION

As in any other engineering discipline, software engineering also has some structured models for software development. This unit will provide you with a generic overview about different software development methodologies adopted by contemporary software firms.

Like any other set of engineering products, software products are also oriented towards the customer. It is either market driven or it drives the market. **Customer Satisfaction** was the buzzword of the 80's. **Customer Delight** in the 90's and **Customer Ecstasy** in the new millennium. Products that are **not customer or user friendly have no place in the market** even though they are engineered using the best technology. The interface of the product is as important as the internal technology of the product.

ISO 12207 is an ISO standard for software lifecycle processes. It aims to be the standard that defines all the tasks required for developing and maintaining software.

The ISO 12207 standard establishes a process of lifecycle for software, including processes and activities applied during the acquisition and configuration of the services of the system. Each Process has a set of outcomes associated with it. There are 23 Processes, 95 Activities, 325 Tasks and 224 Outcomes. These ISO details are not for this work.

The standard has the main objective of supplying a common structure so that the buyers, suppliers, developers, maintainers, operators, managers and technicians involved with the software development use a common language. This common language is established in the form of well defined processes. The structure of the standard was intended to be conceived in a flexible, modular way so as to be adaptable to the necessities of whoever uses it. The standard is based on two basic principles: modularity and responsibility. Modularity means processes with minimum coupling and maximum cohesion. Responsibility means to establish a responsibility for each process, facilitating the application of the standard in projects where many people can be legally involved.

The set of processes, activities and tasks can be adapted according to the software project. These processes are classified in three types: basic, support and organizational. The support and organizational processes must exist independently of the organization and the project being executed. The basic processes are instantiated according to the situation.

2.0 OBJECTIVE

By the end of this unit the student is expected to be conversant with:

i. Generic software development methodologies
ii. Software development life cycle process
iii. Activities involved in initiating a software development project
iv. Software supply and development

3.0 SOFTWARE DEVELOPMENT LIFECYCLE PROCESSES

Software development lifecycle processes contain the core processes involved in creating a software product. We are going to consider software development lifecycle on two perspectives.

1. A software user company wishing to procure new software.
2. A software development company wishing produce new software.

3.1 A SOFTWARE USER COMPANY WISHING TO PROCURE NEW SOFTWARE.

A software user would of a necessity diligently follow five processes to procure a standard software. They are as follows:

i. Acquisition ii. Supply iii. Development iv. Operation v. Maintenance

Because the lifecycle processes cover a very large area a scope was defined. We will explain all the processes briefly but Acquisition and Development more extensively. Each phase within the lifecycle processes can be divided into different activities. This unit explains the different activities for each lifecycle process.

Activity A/ Self Assessment Exercise

1. Explain the implication of ISO 12207 in software development
2. Explain what you mean by software development life cycle
3. Describe the two approaches to software development life cycle

3.2 ACQUISITION

Acquisition covers the activities involved in initiating a project. The acquisition phase can be divided into different activities and deliverables that are completed chronologically.

Initiation: during this activity the following tasks are completed.

i. Define System requirements and obtain approval if applicable;
ii. Define the general requirements of the software
iii. Evaluate other options such as a purchase of an off-the-shelf product or enhancement of an existing product;
iv. If an off-the-shelf product is purchased, the software requirements of this product need to be analyzed.
v. Develop an acquisition plan, which can also be used during the acquisition phase
vi. Define acceptance criteria.
vii. Prepare Request for proposal: during this activity the following tasks are completed:
    a. Acquisition requirements, like System requirements and technical constraints such as target environment, are defined.
    b. Contract milestones for reviewing and supplier's progress audits are defined.
viii. Prepare Contract: during this activity the following tasks are completed
    a. Selection procedure for suppliers are developed;
    b. Suppliers, based on the developed selection procedure, are selected;
    c. The tailor-made ISO/IEC 12207 standard must be included in the contract;
ix. Negotiate changes: Negotiations are held with the selected suppliers;
x. Update contract: Contract is updated with the result from the negotiations in the previous activity.
xi. Supplier monitoring: during this activity the following tasks are completed
    a. Activities of the suppliers according to the agreements made are monitored;
    b. Work together with suppliers to guarantee timely delivery if needed.
xii. Acceptance and completion: during this activity the following tasks are completed
    a. Acceptance tests and procedures are developed;
    b. Acceptance and testing on the product is conducted;
    c. Configuration management on the delivered product is conducted;

3.3 Supply

During the supply phase a project management plan is developed. This plan contains information about the project such as different milestones that need to be reached. This project management plan is needed during the next phase which is the development phase.

3.4 Development

During the development phase the software product is designed, created and tested and will result in a software product ready to be sold to the customer. Throughout time many people have developed means of developing a software application. The choice of developing method often depends on the present situation. The development method which is used in many projects is the V-model. Techniques that can be used during the development are UML for designing and TMap for testing. The following are the most important steps of the V- model.

i. Define software requirements: Gather the software requirements, or demands, for the product that is to be created.
ii. Create High level design: In this phase, the software development process, the software's overall structure and its nuances are defined. In terms of the client/server technology, the number of tiers needed for the package architecture, the database design, the data structure design etc. are all defined in this phase. A software development model is thus created.
iii. **Analysis and Design are very crucial** in the whole development cycle. Any glitch in the design phase could be **very expensive to solve in the later stage** of the software development. Much care is taken during this phase. The logical system of the product is developed in this phase.

   A basic layout of the product is created. This means the setup of different modules and how they communicate with each other. This design does not contain very much detail about the modules.

   The different modules present in the High level design are designed separately. The modules are designed in as much detail as possible.

3.5 CODING

The code is created according to the high level design and the module design.

The design must be translated into a machine-readable form. The code generation step performs this task. If the design is performed in a detailed manner, code generation can be accomplished without much complication. Programming tools like **compilers, interpreters, debuggers** etc... are used to generate the code. Different high level programming languages like **C, C++, Pascal, Java** are used for coding. With respect to the type of application, the right programming language is chosen.

3.6 TESTING

Once the code is generated, the software program testing begins. Different testing methodologies are available to unravel the bugs that were committed during the previous phases. Different testing tools and methodologies are already available. Some companies build their own testing tools that are tailor made for their own development operations.

Activity B/self Assessment Exercise

1. Describe the following concepts with regards to a software user intending to procure a software:

i. software maintenance
ii. software acquisition
iii. software operation

2. What are the steps a software development company would take in producing a software solution for a client?

3.6.1 Execute Module test

The different modules are tested for correct functioning. If this is the case the project can move to the next activity, else the project returns to the module design phase to correct any errors.

3.6.2 Execute Integration test

The communication between modules is tested for correct functioning. If this is the case the project can move to the next activity, else the project falls back to the high level design to correct any errors.

3.6.3 Execute System test

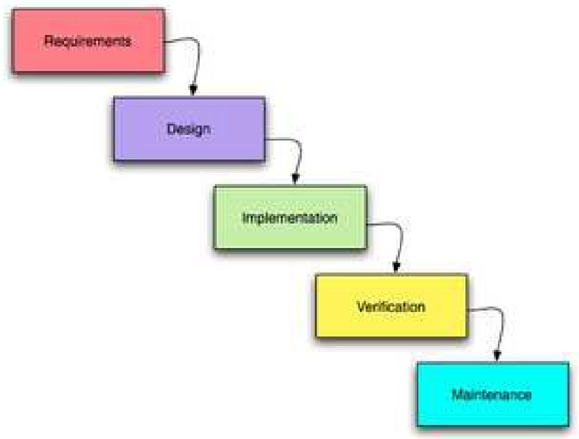
This test checks whether all software requirements are present in the product. If this is the case the product is completed and the product is ready to be transferred to the customer. Else the project falls back to the software requirements activity and the software requirements have to be adjusted.

3.6.4 Maintenance

The software will definitely undergo change once it is delivered to the customer. There can be many reasons for this change to occur. Change could happen because of some unexpected input values into the system. In addition, the changes in the system could directly affect the software operations. The software should be developed to accommodate changes that could happen during the post implementation period.

3.7 Software Development Activities

The activities of the software development process represented in the waterfall model. There are several other models to represent this process.



Software Development Activities from wikipaedia

3.7.1 Planning

The 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.

Once the general requirements are gleaned 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 externally, 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.

3.7.2 Design

Domain Analysis 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 whole new system. Assuming that the developers (including 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 later try to elicit and gather 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 to extracting and gathering the requirements. If an analyst hasn't done the appropriate work confusion may ensue.

3.7.3 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.

3.7.4 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 step also addresses interfaces between the software system and other software products, as well as the underlying hardware or the host operating system.

3.7.5 Implementation, testing and documenting

Implementation is the part of the process where software engineers actually program the code for the project.

Software testing is an integral and important part of the software development process. This part 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.

3.7.6 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 cope with newly discovered problems or new requirements can take far more time than the initial development of the software. It may be necessary to add 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. If the labor cost of the maintenance phase exceeds 25% of the prior-phases' labor cost, then it is likely that the overall quality, of at least one prior phase, is poor. In that case, management should consider the option of rebuilding the system (or portions) before maintenance cost is out of control.

Bug Tracking System tools are often deployed at this stage of the process to allow development teams to interface with customer/field teams testing the software to identify any real or perceived issues. These software tools, both open source and commercially licensed, provide a customizable process to acquire, review, acknowledge, and respond to reported issues.

Tutor Marked assignment

1. Differentiate between executive module test and executive integration test
2. Explain what you understand by software deployment
3. Describe in details what you mean by bugs

4.0 Summary and Conclusion

Like any other set of engineering products, software products are also oriented towards the customer. It is either market driven or it drives the market. A software project has an entry and exit point. It is paramount to note that the software products has to be maintained and updated to keep abreast with the corporate expansion.

5.0 References and Further Reading

1. Roger S. Pressman, *Software Engineering (A practitioner's approach)*, 5th edition, 2000, Mc Graw-Hill Education
2. Government Accountability Report (January 2003). Report GAO-03-343, National Airspace System: Better Cost Data Could Improve FAA’s Management of the Standard Terminal Automation Replacement System. Retrieved from http://www.gao.gov/cgi-bin/getrpt?GAO-03-343
3. Gerhard Fischer, "The Software Technology of the 21st Century: From Software Reuse to Collaborative Software Design", 2001
4. Lydia Ash: *The Web Testing Companion: The Insider's Guide to Efficient and Effective Tests*, Wiley, May 2, 2003.
5. www.SaaSSDLC.com - Software as a Service Systems Development Life Cycle Project
6. Software development life cycle (SDLC) [visual image], *software development life cycle*
7. Iterative Development and The Leaning Tower of Pisa - From The Trench
8. Selecting an SDLC", 2009

Contents

1.0 Introduction

Software systems come and go through a series of passages that account for their inception, initial development, productive operation, upkeep, and retirement from one generation to another. This unit categorizes and examines a number of methods for describing or modeling how software systems are developed. It begins with background and definitions of traditional software life cycle models that dominate most current software development practices. This is followed by a more comprehensive review of the alternative models of software evolution that are of current use as the basis for organizing software engineering projects and technologies.

2.0 objectives

By the end of this unit the student is expected to understand:

i. what software development life cycle is
ii. how to differentiate between the different software development models
iii. what software production model is
iv. how to apply these models in software development

3.0 History

Explicit models of software evolution date back to the earliest projects developing large software
systems in the 1950's and 1960's. Overall, the apparent purpose of these early software life cycle models was to provide a conceptual scheme for rationally managing the development of software systems. Such a scheme could therefore serve as a basis for planning, organizing, staffing, coordinating, budgeting, and directing software development activities.

Since the 1960's, many descriptions of the classic software life cycle have appeared. Royce in 1970 originated the formulation of the software life cycle using the now familiar "waterfall" chart, displayed in Figure 1. The chart summarizes in a single display how developing large software systems are difficult because it involves complex engineering tasks that may require iteration and rework before completion. These charts are often employed during introductory presentations, for people (e.g., customers of custom software) who may be unfamiliar with the various technical problems and strategies that must be addressed when constructing large software systems. These classic software life cycle models usually include some version or subset of the following activities:

i. *System Initiation/Planning:* where do systems come from? In most situations, new feasible systems replace or supplement existing information processing mechanisms whether they were previously automated, manual, or informal.

ii. *Requirement Analysis and Specification:* identifies the problems a new software system is suppose to solve, its operational capabilities, its desired performance characteristics, and the resource infrastructure needed to support system operation and maintenance.

iii. *Functional Specification or Prototyping*: identifies and potentially formalizes the objects of computation, their attributes and relationships, the operations that transform these objects, the constraints that restrict system behaviour, and so forth.

iv. *Partition and Selection* (Build vs. Buy vs. Reuse): given requirements and functional specifications, divide the system into manageable pieces that denote logical subsystems, then determine whether new, existing, or reusable software systems correspond to the needed pieces.

v. *Architectural Design and Configuration Specification*: defines the interconnection and resource interfaces between system subsystems, components, and modules in ways suitable for their detailed design and overall configuration management.

vi. *Detailed Component Design Specification*: defines the procedural methods through which the data resources within the modules of a component are transformed from required inputs into provided outputs.

vii. *Component Implementation and Debugging*: codifies the preceding specifications into operational source code implementations and validates their basic operation.

viii. *Software Integration and Testing*: affirms and sustains the overall integrity of the software system architectural configuration through verifying the consistency and completeness of implemented modules, verifying the resource interfaces and interconnections against their specifications, and validating the performance of the system and subsystems against their requirements.

ix. *Documentation Revision and System Delivery*: packaging and rationalizing recorded system development descriptions into systematic documents and user guides, all in a form suitable for dissemination and system support.

x. *Deployment and Installation*: providing directions for installing the delivered software into the local computing environment, configuring operating systems parameters and user access privileges, and running diagnostic test cases to assure the viability of basic system operation.

xi. *Training and Us*e: providing system users with instructional aids and guidance for understanding the system's capabilities and limits in order to effectively use the system.

xii. *Software Maintenance*: sustaining the useful operation of a system in its host/target environment by providing requested functional enhancements, repairs, performance improvements, and conversions.

3.1 What is a software life cycle model?

A software life cycle model is either a descriptive or prescriptive characterization of how software is or should be developed. A descriptive model describes the history of how a particular
software system was developed. Descriptive models may be used as the basis for understanding
and improving software development processes, or for building empirically grounded prescriptive models. A prescriptive model prescribes how a new software system should be developed. Prescriptive models are used as guidelines or frameworks to organize and structure how software development activities should be performed, and in what order. Typically, it is easier and more common to articulate a prescriptive life cycle model for how software systems should be developed. This is possible since most such models are intuitive or well reasoned. This means that many idiosyncratic details that describe how a software system is built in practice can be ignored, generalized, or deferred for later consideration. This, of course, should raise concern for the relative validity and robustness of such life cycle models when developing different kinds of application systems, in different kinds of development settings, using different programming languages, with differentially skilled staff, etc. However, prescriptive models are also used to package the development tasks and

techniques for using a given set of software engineering tools or environment during a development project.

Descriptive life cycle models, on the other hand, characterize how particular software systems
are actually developed in specific settings. As such, they are less common and more difficult to
articulate for an obvious reason: one must observe or collect data throughout the life cycle of a
software system, a period of elapsed time often measured in years. Also, descriptive models are
specific to the systems observed and only generalizable through systematic comparative analysis.
Therefore, this suggests the prescriptive software life cycle models will dominate attention until
a sufficient base of observational data is available to articulate empirically grounded descriptive
life cycle models.
These two characterizations suggest that there are a variety of purposes for articulating software
life cycle models. These characterizations serve as a

i. Guideline to organize, plan, staff, budget, schedule and manage software project work over organizational time, space, and computing environments.
ii. Prescriptive outline for what documents to produce for delivery to client.
iii. Basis for determining what software engineering tools and methodologies will be most appropriate to support different life cycle activities.
iv. Framework for analyzing or estimating patterns of resource allocation and consumption during the software life cycle
v. Basis for conducting empirical studies to determine what affects software productivity, cost, and overall quality.

3.2 What is a software process model?

In contrast to software life cycle models, software process models often represent a networked sequence of activities, objects, transformations, and events that embody strategies for accomplishing software evolution. Such models can be used to develop more precise and formalized descriptions of software life cycle activities. Their power emerges from their utilization of a sufficiently rich notation, syntax, or semantics, often suitable for computational processing.
Software process networks can be viewed as representing multiple interconnected task chains
Task chains represent a non-linear sequence of actions that structure and transform available computational objects (resources) into intermediate or finished products. Non-linearity implies that the sequence of actions may be non-deterministic, iterative, accommodate multiple/parallel alternatives, as well as partially ordered to account for incremental progress.
Task actions in turn can be viewed a non-linear sequences of primitive actions which denote atomic units of computing work, such as a user's selection of a command or menu entry using a mouse or keyboard. These units of cooperative work between people and computers are referred to as structured discourses of work, while task chains have become popularized under the name of workflow.

Task chains can be employed to characterize either prescriptive or descriptive action sequences.
Prescriptive task chains are idealized plans of what actions should be accomplished, and in what
order. For example, a task chain for the activity of object-oriented software design might include
the following task actions:

i. Develop an informal narrative specification of the system.
ii. Identify the objects and their attributes.
iii. Identify the operations on the objects.
iv. Identify the interfaces between objects, attributes, or operations.
v. Implement the operations.

Clearly, this sequence of actions could entail multiple iterations and non-procedural primitive action invocations in the course of incrementally progressing toward an object-oriented software design.
Task chains join or split into other task chains resulting in an overall production network or web. The production web represents the organizational production system that transforms raw computational, cognitive, and other organizational resources into assembled, integrated and usable software systems. The production lattice therefore structures how a software system is developed, used, and maintained. However, prescriptive task chains and actions cannot be formally guaranteed to anticipate all possible circumstances or idiosyncratic foul-ups that can emerge in the real world of software development. Thus, any software production web will in some way realize only an approximate or incomplete description of software development.

Articulation work is a kind of unanticipated task that is performed when a planned task chain is
inadequate or breaks down. It is work that represents an open-ended non-deterministic sequence
of actions taken to restore progress on the disarticulated task chain, or else to shift the flow of productive work onto some other task chain. Thus, descriptive task chains are employed to characterize the observed course of events and situations that emerge when people try to follow a planned task sequence.
Articulation work in the context of software evolution includes actions people take that entail either their accommodation to the contingent or anomalous behaviour of a software system, or
negotiation with others who may be able to affect a system modification or otherwise alter current circumstances.
This notion of articulation work has also been referred to as software process dynamism.

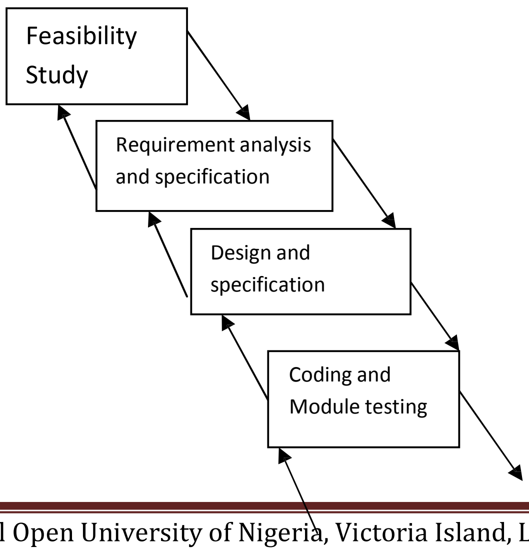
3.3 Traditional Software Life Cycle Models

Traditional models of software evolution have been with us since the earliest days of software engineering. In this section, we identify four. The classic software life cycle (or "waterfall chart")
and stepwise refinement models are widely instantiated in just about all books on modern programming practices and software engineering. The incremental release model is closely related to industrial practices where it most often occurs. Military standards based models have
also certain forms of the classic life cycle model into required practice for government contractors. Each of these four models uses coarse-grain or macroscopic characterizations when

describing software evolution. The progressive steps of software evolution are often described as
stages, such as requirements specification, preliminary design, and implementation; these usually
have little or no further characterization other than a list of attributes that the product of such a
stage should possess. Further, these models are independent of any organizational development
setting, choice of programming language, software application domain, etc. In short, the traditional models are context-free rather than context-sensitive. But as all of these life cycle models have been in use for some time, we refer to them as the traditional models, and characterize each in turn.

3.4 Classic Software Life Cycle

The classic software life cycle is often represented as a simple prescriptive waterfall software phase model, where software evolution proceeds through an orderly sequence of transitions from
one phase to the next in order. Such models resemble finite state machine descriptions of software evolution. However, these models have been perhaps most useful in helping to structure, staff, and manage large software development projects in complex organizational settings, which was one of the primary purposes.
Alternatively, these classic models have been widely characterized as both poor descriptive and
prescriptive models of how software development "in-the-small" or "in-the-large" can or should
occur. Figure 1 provides a common view of the waterfall model for software development attributed to Royce.


Feasibility Study
Requirement analysis and specification
Design and specification
Coding and Module testing

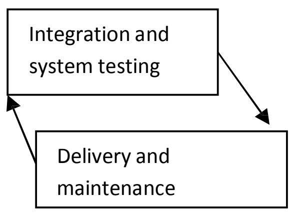


Figure 1. The Waterfall Model of Software Development (Royce 1970)

Activity A/Self Assessment Exercise

1. What are the steps involved in classic software development
2. Explain the following terms:
   i. Software life cycle model
   ii. Prescriptive model
   iii. Descriptive model
3. What is software process model

3.4.1 Stepwise Refinement

In this approach, software systems are developed through the progressive refinement and enhancement of high-level system specifications into source code components. However, the choice and order of which steps to choose and which refinements to apply remain unstated. Instead, formalization is expected to emerge within the heuristics and skills that are acquired and applied through increasingly competent practice. This model has been most effective and widely applied in helping to teach individual programmers how to organize their software development work. Many interpretations of the classic software life cycle thus subsume this approach within their design and implementations.

3.4.2 Incremental Development and Release

Developing systems through incremental release requires first providing essential operating functions, then providing system users with improved and more capable versions of a system at
regular intervals. This model combines the classic software life cycle with iterative enhancement at the level of system development organization. It also supports a strategy to periodically distribute software maintenance updates and services to dispersed user communities.
This in turn accommodates the provision of standard software maintenance contracts. It is therefore a popular model of software evolution used by many commercial software firms and
system vendors. This approach has also been extended through the use of software prototyping
tools and techniques, which more directly provide support for incremental development and iterative release for early and ongoing user feedback and evaluation.

Figure 2 provides an example view of an incremental development, build, and release model for engineering large Ada-based software systems, developed by Royce in 1990 at TRW. Elsewhere, the Cleanroom software development method at use in IBM and NASA laboratories
provides incremental release of software functions and/or subsystems (developed through

stepwise refinement) to separate in-house quality assurance teams that apply statistical measures
and analyses as the basis for certifying high-quality software systems.

3.4.3 Industrial and Military Standards, and Capability Models

Industrial firms often adopt some variation of the classic model as the basis for standardizing their software development practice. Such standardization is often motivated by needs to simplify or eliminate complications that emerge during large software development or project management.

From the 1970's through the present, many government contractors organized their software development activities according to succession of military software standards such as MIL-STD-
2167A, MIL-STD 498, and IEEE-STD-016. ISO12207 is now the standard that most such contractors now follow. These standards are an outgrowth of the classic life cycle activities, together with the documents required by clients who procure either software systems or complex platforms with embedded software systems. Military software system are often constrained in ways not found in industrial or academic practice, including:

(1) Required use of military standard computing equipment (which is often technologically dated and possesses limited processing capabilities);

(2) Are embedded in larger systems (e.g., airplanes, submarines, missiles, command and control systems) which are mission-critical (i.e., those whose untimely failure could result in military disadvantage and/or life-threatening risks);

(3) Are developed under contract to private firms through cumbersome procurement and acquisition procedures that can be subject to public scrutiny and legislative intervention;

(4) Many embedded software systems for the military are among the largest and most complex systems in the world.

Finally, the development of custom software systems using commercial off-the-shelf (COTS) components or products is a recent direction for government contractors, and thus represents new challenges for how to incorporate a component-based development into the overall software life cycle. Accordingly, new software life cycle models that exploit COTS components will continue to appear in the next few years.

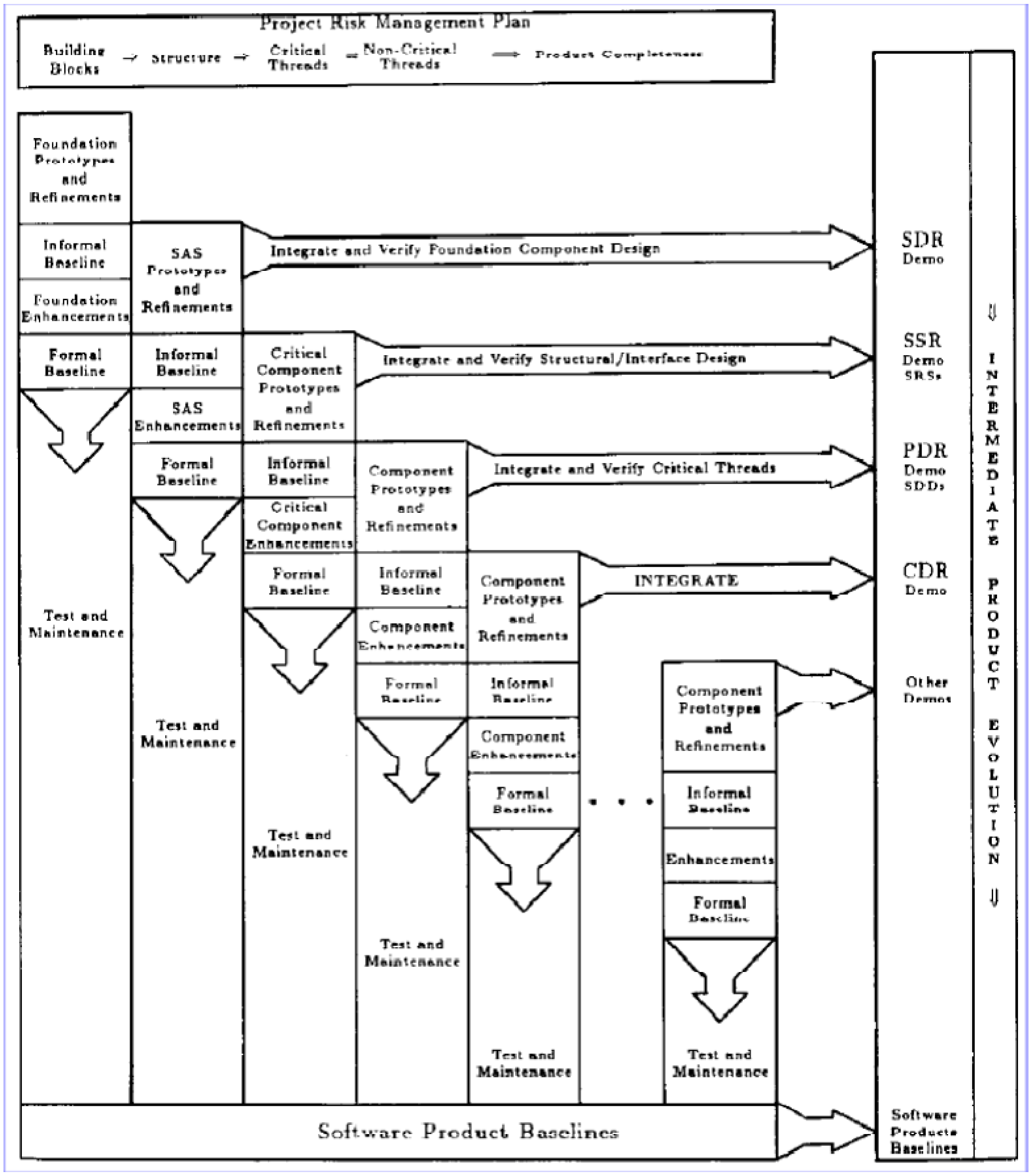

Project Risk Management Plan
Building Blocks → Structure → Critical Threads → Non-Critical Threads → Product Completeness
Foundation Prototypes and Refinements
Informal Baseline
Foundation Enhancements
Formal Baseline
SAS Prototypes and Refinements
Informal Baseline
SAS Enhancements
Formal Baseline
Critical Component Prototypes and Refinements
Informal Baseline
Critical Component Enhancements
Formal Baseline
Component Prototypes and Refinements
Informal Baseline
Component Enhancements
Formal Baseline
Component Prototypes and Refinements
Informal Baseline
Component Enhancements
Formal Baseline
Component Prototypes and Refinements
Informal Baseline
Enhancements
Formal Baseline
Test and Maintenance
Integrate and Verify Foundation Component Design
Integrate and Verify Structural/Interface Design
Integrate and Verify Critical Threads
INTEGRATE
SDR Demo
SSR Demo SRSs
PDR Demo SDDs
CDR Demo
Other Demos
Software Product Baselines
Software Products Baselines
INTERMEDIATE PRODUCT EVOLUTION

Figure 2. An Incremental Development, Build, and Release Model (Royce 1990)

In industrial settings, standard software development models represent often provide explicit detailed guidelines for how to deploy, install, customize or tune a new software system release in
its operating application environment. In addition, these standards are intended to be compatible
with provision of software quality assurance, configuration management, and independent verification and validation services in a multi-contractor development project.

These efforts in turn help pave the way for what many software development organizations now practice, or have been certified to practice, software process capability assessments, following the Capability Maturity Model developed by the Software Engineering Institute .

3.5 Alternatives to the Traditional Software Life Cycle Models

There are at least three alternative sets of models of software development. These models are alternatives to the traditional software life cycle models. These three sets focus attention to either the products, production processes, or production settings associated with software development. Collectively, these alternative models are finer-grained, often detailed to the point
of computational formalization, more often empirically grounded, and in some cases address the
role of new automated technologies in facilitating software development. We examine each set of models in the following sections.

3.5.1 Software Product Development Models

Software products represent the information-intensive artifacts that are incrementally constructed
and iteratively revised through a software development effort. Such efforts can be modeled using
software product life cycle models. These product development models represent an evolutionary
revision to the traditional software life cycle models. The revisions arose due to the availability of new software development technologies such as software prototyping languages and environments, reusable software, application generators, and documentation support environments. Each of these technologies seeks to enable the creation of executable software implementations either earlier in the software development effort or more rapidly. Therefore in this regard, the models of software development may be implicit in the use of the
technology, rather than explicitly articulated. This is possible because such models become increasingly intuitive to those developers whose favourable experiences with these technologies
substantiate their use. Thus, detailed examination of these models is most appropriate when such
technologies are available for use or experimentation.

3.5.2 Rapid Prototyping and Joint Application Development

Prototyping is a technique for providing a reduced functionality or a limited performance version
of a software system early in its development. In contrast to the classic system life cycle, prototyping is an approach whereby more emphasis, activity, and processing are directed to the early stages of software development (requirements analysis and functional specification). In turn, prototyping can more directly accommodate early 10 user participation in determining, shaping, or evaluating emerging system functionality.
Therefore, these up-front concentrations of effort, together with the use of prototyping technologies, seeks to trade-off or otherwise reduce downstream software design activities and
iterations, as well as simplify the software implementation effort.
Software prototypes come in different forms including throwaway prototypes, mock-ups,

demonstration systems, quick-and-dirty prototypes, and incremental evolutionary prototypes. Increasing functionality and subsequent evolvability is what distinguishes the prototype forms on this list.
Prototyping technologies usually take some form of software functional specifications as their
starting point or input, which in turn is simulated, analyzed, or directly executed. These technologies can allow developers to rapidly construct early or primitive versions of software systems that users can evaluate. User evaluations can then be incorporated as feedback to refine
the emerging system specifications and designs. Further, depending on the prototyping technology, the complete working system can be developed through a continual revising/refining
the input specifications. This has the advantage of always providing a working version of the emerging system, while redefining software design and testing activities to input specification refinement and execution. Alternatively, other prototyping approaches are best suited for developing throwaway or demonstration systems, or for building prototypes by reusing part/all
of some existing software systems. Subsequently, it becomes clear why modern models of software development like the Spiral Model and the ISO 12207 expect that prototyping will be a common activity that facilitates the capture and refinement of software requirements, as well as overall software development.

3.5.3 Joint Application Development (JAD)
This is a technique for engaging a group or team of software developers, testers, customers, and prospective end-users in a collaborative requirements elicitation and prototyping effort. JAD is quintessentially a technique for facilitating group interaction and collaboration. Consultants often employ JAD or external software system vendors who have been engaged to build a custom software system for use in a particular organizational setting. The JAD process is based on four ideas:
1.People who actually work at a job have the best understanding of that job.
2.People who are trained in software development have the best understanding of the possibilities of that technology.
3. Software-based information systems and business processes rarely exist in isolation -- they transcend the confines of any single system or office and effect work in related departments.
People working in these related areas have valuable insight on the role of a system within a larger community.
4.The best information systems are designed when all of these groups work together on a project as equal partners.

Following these ideas, it should be possible for JAD to cover the complete development life cycle of a system. The JAD is usually a 3 to 6 month well-defined project, when systems can be
constructed from commercially available software products that do not require extensive coding
or complex systems integration. For large-scale projects, it is recommended that the project be organized as an incremental development effort, and that separate JAD's be used for each increment. Given this formulation, it is possible to view open source software development projects that rely on group email discussions among globally distributed users and

developers, together with Internet-based synchronized version updates as an informal variant of JAD.

3.5.4 Assembling Reusable Components

The basic approach of reusability is to configure and specialize pre-existing software components into viable application systems. Such source code components might already have associated specifications and designs associated with their implementations, as well as have been tested and certified. However, it is also clear that software domain models, system specifications, designs, test case suites, and other software abstractions may themselves be treated as reusable software development components. These components may have a greater potential for favourable impact on reuse and semi-automated system generation or composition. Therefore,
assembling reusable software components is a strategy for decreasing software development effort in ways that are compatible with the traditional life cycle models.
The basic dilemmas encountered with reusable software componentry include:
(a) acquiring,analyzing and modeling a software application domain,
(b) how to define an appropriate software part naming or classification scheme,
(c) collecting or building reusable software components,
(d) configuring or composing components into a viable application,
(e) maintaining and searching a components library.
In turn, each of these dilemmas is mitigated or resolved in practice through the selection of software component granularity. The granularity of the components (i.e., size, complexity, and functional capability) varies greatly across different approaches. Most approaches attempt to utilize components similar to common data structures with algorithms for their manipulation: small-grain components. However, the use/reuse of small-grain components in and of itself does not constitute a distinct approach to software development.

Other approaches attempt to utilize components resembling functionally complete systems or subsystems (e.g., user interface management system): large-grain components. The use/reuse of large-grain components guided by an application domain analysis and subsequent mapping of attributed domain objects and operations onto interrelated components does appear to be an alternative approach to developing software systems.

There are many ways to utilize reusable software components in evolving software systems such as during architectural or component design specification as a way to speed implementation. They might also be used for prototyping purposes if a suitable software prototyping technology is available.

3.5.5 Application Generation

Application generation is an approach to software development similar to reuse of parameterized, large-grain software source code components. Such components are configured and specialized to an application domain via a formalized specification language used as input to the application generator. Common examples provide standardized interfaces to database management system applications, and include generators for reports, graphics, user interfaces, and application-specific editors.

Application generators give rise to a model of software development whereby traditional software design activities are either all but eliminated, or reduced to a data base design problem.
The software design activities are eliminated or reduced because the application generator

embodies or provides a generic software design that should be compatible with the application
domain. However, users of application generators are usually expected to provide input specifications and application maintenance services. These capabilities are possible since the generators can usually only produce software systems specific to a small number of similar application domains, and usually those that depend on a data base management system.

3.5.6 Software Documentation Support Environments

Much of the focus on developing software products draws attention to the tangible software artifacts that result. Most often, these products take the form of documents: commented source code listings, structured design diagrams, unit development folders, etc. These documents characterize what the developed system is suppose to do, how it does it, how it was developed, how it was put together and validated, and how to install, use, and maintain it. Thus, a collection of software documents records the passage of a developed software system through a set of life
cycle stages. It seems reasonable that there will be models of software development that focus attention to the systematic production, organization, and management of the software development documents.
Further, as documents are tangible products, it is common practice when software systems are
developed under contract to a private firm, that the delivery of these documents is a contractual
stipulation, as well as the basis for receiving payment for development work already performed.
Thus, the need to support and validate conformance of these documents to software development
and quality assurance standards emerges. However, software development documents are often a
primary medium for communication between developers, users, and maintainers that spans organizational space and time. Thus, each of these groups can benefit from automated mechanisms that allow them to browse, query, retrieve, and selectively print documents. As such, we should not be surprise to see construction and deployment of software environments that provide ever increasing automated support for engineering the software documentation life cycle, or how these capabilities have since become part of the commonly available computeraided software engineering (CASE) tools suites like Rational Rose, and others based on the use of the Unified Modeling Language (UML).

3.5.7Rapid Iteration, Incremental Evolution, and Evolutionary Delivery

There are a growing number of technological, social and economic trends that are shaping how a
new generation of software systems are being developed that exploit the Internet and World Wide Web. These include the synchronize and stabilize techniques popularized by Microsoft and
Netscape at the height of the fiercely competitive efforts to dominate the Web browser market of
the mid 1990's. They also include the development of open source software systems that rely on a decentralized community of volunteer software developers to collectively develop and test software systems that are incrementally enhanced, released, experienced, and debugged in an overall iterative and cyclic manner. The elapsed time of these incremental development life cycles on some projects may be measured in weeks, days, or hours! The centralized

planning, management authority and coordination imposed by the traditional system life cycle model has been discarded in these efforts, replaced instead by a more organic, participatory, reputation-based, and community oriented engineering practice.

Software engineering in the style of rapid iteration and incremental evolution is one that focuses on and celebrates the inevitability of constantly shifting system requirements, unanticipated situations of use and functional enhancement, and the need for developers to collaborate with one another, even when they have never met.

3.6 Program Evolution Models

In contrast to the preceding four prescriptive product development models, Lehman and Belady
sought to develop a descriptive model of software product evolution at IBM.
Based on their investigations, they identify five properties that characterize the evolution of large software systems. These are:

1. *Continuing change*: a large software system undergoes continuing change or becomes progressively less useful
2. *Increasing complexity*: as a software system evolves, its complexity increases unless work is done to maintain or reduce it
3. *Fundamental law of program evolution*: program evolution, the programming process, and global measures of project and system attributes are statistically self-regulating with determinable trends and invariances
4. *Invariant work rate*: the rate of global activity in a large software project is statistically invariant
5. *Incremental growth limit*: during the active life of a large program, the volume of modifications made to successive releases is statistically invariant.

However, note that these are global properties of large software systems and not causal mechanisms of software development.

Activity B/Self Assessment Exercise

1. Describe the stepwise refinement approach
2. List the features that differentiates military software from industrial software
3. Describe the following concepts
   i. Rapid prototyping
   ii. Joint application development

3.7 Software Production Process Models

There are two kinds of software production process models: non-operational and operational. Both are software process models. The difference between the two primarily stems from the fact
that the operational models can be viewed as computational scripts or programs: programs that
implement a particular regimen of software engineering and development. Non-operational models on the other hand denote conceptual approaches that have not yet been sufficiently articulated in a form suitable for codification or automated processing.

3.7.1 Non-Operational Process Models

There are two classes of non-operational software process models of interest. These are the spiral model and the continuous transformation models. There is also a wide selection of other non-operational models, which for brevity we label as miscellaneous models. Each is examined in turn.

3.7.1.1The Spiral Model.

The spiral model of software development and evolution represents a risk driven approach to software process analysis and structuring. This approach, developed by Barry Boehm, incorporates elements of specification-driven, prototype-driven process methods, together with the classic software life cycle. It does so by representing iterative development cycles as an expanding spiral, with inner cycles denoting early system analysis and prototyping, and outer cycles denoting the classic software life cycle.
The radial dimension denotes cumulative development costs, and the angular dimension denotes
progress made in accomplishing each development spiral. See Figure 3.
Risk analysis, which seeks to identify situations that might cause a development effort to fail or
go over budget/schedule, occurs during each spiral cycle. In each cycle, it represents roughly the
same amount of angular displacement, while the displaced sweep volume denotes increasing levels of effort required for risk analysis. System development in this model therefore spirals out
only so far as needed according to the risk that must be managed. Alternatively, the spiral model
indicates that the classic software life cycle model need only be followed when risks are greatest,
and after early system prototyping as a way of reducing these risks, albeit at increased cost. The
insights that the Spiral Model offered has in turn influenced the standard software life cycle process models, such as ISO12207 noted earlier. Finally, efforts are now in progress to integrate
computer-based support for stakeholder negotiations and capture of trade-off rationales into an
operational form of the WinWin Spiral Model

3.7.1.2 Miscellaneous Process Models.

Many variations of the non-operational life cycle and process models are available. These include fully interconnected life cycle models that accommodate transitions between any two phases subject to satisfaction of their pre- and post-conditions, as well as compound variations on the traditional life cycle and continuous transformation models. However, reports indicate that in general most software process models are exploratory, though there is now a growing base of experimental or industrial experience with these models.

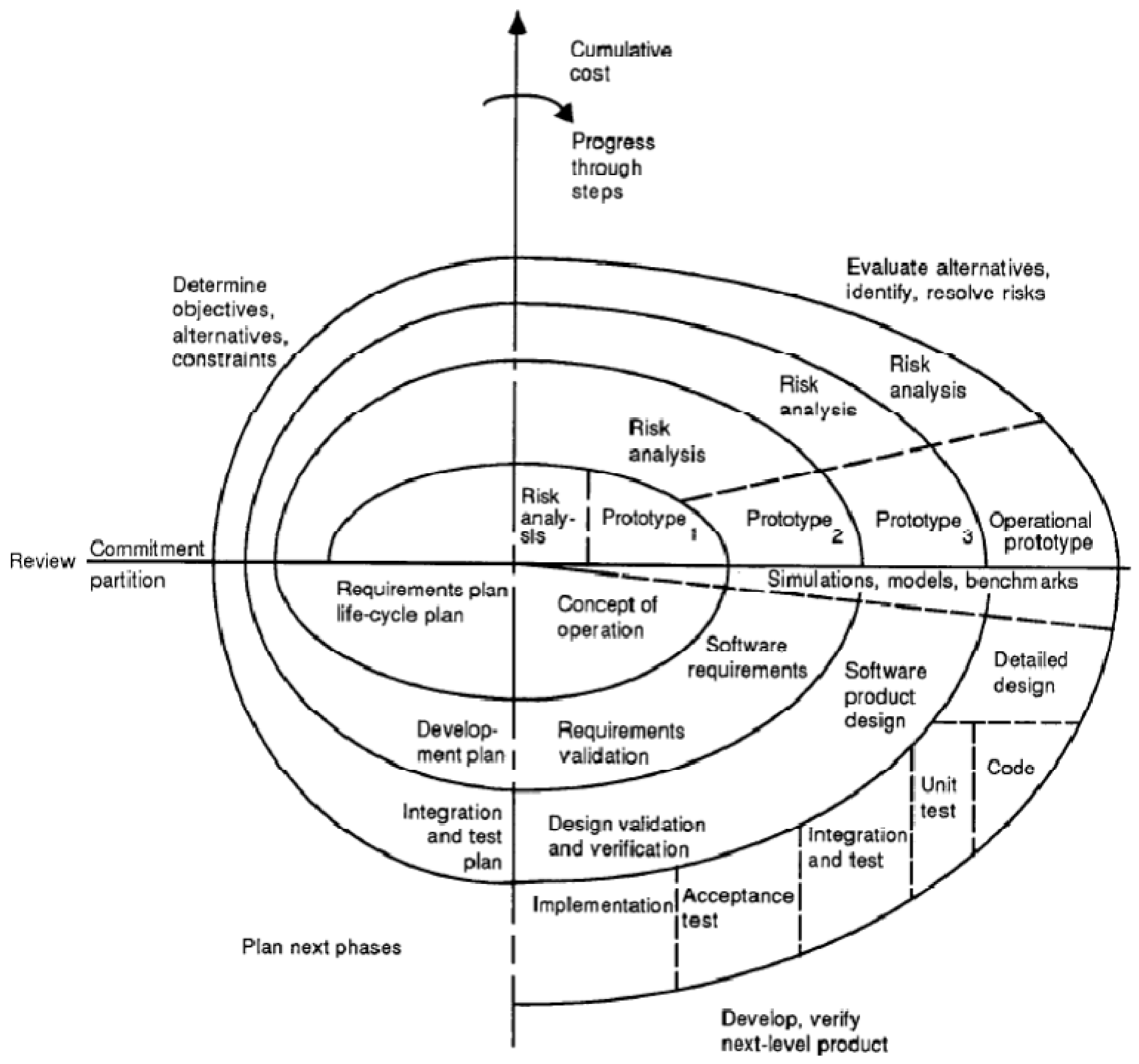


Figure 3.The Spiral Model diagram from (Boehm 1987)

3.8 Operational Process Models

In contrast to the preceding non-operational process models, many models are now beginning to
appear which codify software engineering processes in computational terms--as programs or executable models. Three classes of operational software process models can be identified and
examined. Following this, we can also identify a number of emerging trends that exploit and extend the use of operational process models for software engineering.

3.8.1 Operational specifications for rapid prototyping.

The operational approach to software development assumes the existence of a formal specification language and processing environment that supports the evolutionary development of specifications into an prototype implementation. Specifications in the language are coded, and when computationally evaluated, constitute a functional prototype of the specified system. When such specifications can be developed and processed incrementally, the resulting system prototypes can be refined and evolved into functionally more complete systems. However, the emerging software systems are always operational in some form during their development.

Variations within this approach represent either efforts where the prototype is the end sought, or where specified prototypes are kept operational but refined into a complete system.

The specification language determines the power underlying operational specification technology. Simply stated, if the specification language is a conventional programming language, then nothing new in the way of software development is realized. However, if the specification incorporates (or extends to) syntactic and semantic language constructs that are specific to the application domain, which usually are not part of conventional programming languages, then domain-specific rapid prototyping can be supported. An interesting twist worthy of note is that it is generally within the capabilities of many operational specification languages to specify "systems" whose purpose is to serve as a model of an arbitrary abstract process, such as a software process model. In this way, using a prototyping language and environment, one might be able to specify an abstract model of some software engineering processes as a system that produces and consumes certain types of documents, as well as the classes of development transformations applied to them. Thus, in this regard, it may be possible to construct operational software process models that can be executed or simulated using software prototyping technology.

3.8.2 Software automation.

Automated software engineering (also called knowledge-based software engineering) attempts to take process automation to its limits by assuming that process specifications can be used directly to develop software systems, and to configure development environments to support the production tasks at hand. The common approach is to seek to automate some form of the continuous transformation model. In turn, this implies an automated environment capable of recording the formalized development of operational specifications, successively transforming and refining these specifications into an implemented system, assimilating maintenance requests by incorporating the new/enhanced specifications into the current development derivation, then replaying the revised development toward implementation. However, current progress has been limited to demonstrating such mechanisms and specifications on software coding, maintenance, project communication and management tasks, as well as to software component catalogs and formal models of software development processes. Now we can combine different life cycle, product, and production process models within a process-driven framework that integrates both conventional and knowledge-based software engineering tools and environments.

3.8.3 Software process automation and programming.

Process automation and programming are concerned with developing formal specifications of how a system or family of software systems should be developed. Such specifications therefore provide an account for the organization and description of various software production task chains, how they interrelate, when then can iterate, etc., as well as what software tools to use to support different tasks, and how these tools should be used. Focus then converges on characterizing the constructs incorporated into the language for specifying and programming software processes.
Therefore, looking the appropriateness of language constructs for expressing rules for backward and forward-chaining, behaviour, object type structures, process dynamism, constraints, goals, policies, modes of user interaction, plans, off-line activities, resource commitments, etc. across various levels of granularity, we discover that conventional mechanisms such as operating system shell scripts (e.g., Makefiles on Unix) do not support the kinds of software process automation these constructs portend.

Many alternative process model formalisms were tried including knowledge-based representations, rule-based schemes, and Petri-net schemes and variations. In the early 1990's, emphasis focused on the development of distributed client-server environments that generally relied on a centralized server. The server might then interpret a process model for how to schedule, coordinate, or reactively synchronize the software engineering activities of developers working with client-side tools. To no surprise, by the late 1990's emphasis has shifted towards environment architectures that employed decentralized servers for process
support, workflow automation, data storage, and tool services. Finally, there was also some effort to expand the scope of operational support bound to process models in terms that recognized their growing importance as a new kind of software.
Here we began to see the emergence of process engineering environments that support their own class of life cycle activities and support mechanisms.

3.9 Emerging Trends and New Directions

In addition to the ongoing interest, that is, the assessment of process-centered or process-driven
software engineering environments that rely on process models to configure or control their operation, there are a number of promising avenues that should be optimised in software process models. These opportunities areas and sample direction for further exploration include:

i. Software process simulation efforts which seek to determine or experimentally evaluate the performance of classic or operational process models using a sample of alternative parameter configurations or empirically derived process data. Simulation of empirically derived models of software evolution or evolutionary processes also offer new avenues for exploration.
ii. Web-based software process models and process engineering environments that seek to provide software development workspaces and project support capabilities that are tied to adaptive process models.
iii. Software process and business process reengineering which focuses attention to opportunities that emerge when the tools, techniques, and concepts for each disciplined are combined to their relative advantage. This in turn is giving rise to new techniques for redesigning, situating, and optimizing software process models for specific organizational and system development settings
iv. Understanding, capturing, and operationalising process models that characterize the practices and patterns of globally distributed software development associated with open source software, as well as other emerging software development processes, such as extreme programming and Web-based virtual software development enterprises or workspaces.

Tutor Marked Assignment

1. with the aid of a diagram describe spiral model
2. differentiate between process automation and software automation
3. what are the benefits of incremental release

4.0 Summary and Conclusions

The central point of this unit is that contemporary models of software development must account for the interrelationships between software products and production processes, as well as for the roles played by tools, people and their workplaces. Modeling these patterns can utilize features of traditional software life cycle models, as well as those of automatable software process models. Nonetheless, we must also recognize that the death of the traditional system life cycle model may be at hand. New models for software development enabled by the Internet, group facilitation and distant coordination within open source software communities, and shifting business imperatives in response to these conditions are giving rise to a new generation of software processes and process models. These new models provide a view of software development and evolution that is incremental, iterative, ongoing, interactive, and sensitive to social and organizational circumstances, while at the same time, increasingly amenable to automated support, facilitation, and collaboration over the distances of space and time.

5.0 References and Further Reading

1. Ambriola, V., R. Conradi and A. Fuggetta, Assessing process-centered software engineering
environments, *ACM Trans. Software Engineering Methodology*. 1997.
2. Balzer, R., Transformational Implementation: An Example, *IEEE Trans. Software Engineering*.
3. Balzer, R., A 35 Year Perspective on Automatic Programming, *IEEE Trans. Software Engineering*, 2005.
Basili, V.R. and H.D. Rombach, The TAME Project: Towards Improvement-Oriented Software
Environments, *IEEE Trans. Soft. Engr.*, 2008.
Basili, V. R., and A. J. Turner, Iterative Enhancement: A Practical Technique for Software Development, *IEEE Trans. Software Engineering*, 1995.
Batory, D., V. Singhal, J. Thomas, S. Dasari, B. Geraci, M. Sirkin, The GenVoca model of software-system generators, *IEEE Software*, 2004.
Beck, K. *Extreme Programming Explained*, Addison-Wesley, Palo Alto, CA, 1999.
Boehm, B. W., *Software Engineering Economics*, Prentice-Hall, Englewood Cliffs, N. J., 2001
Boehm, B., A Spiral Model of Software Development and Enhancement, 2007.
Boehm, B., A. Egyed, J. Kwan, D. Port, A. Shah, and R. Madachy, Using the WinWin Spiral Model: A Case Study, 2005.
Cook, J.E., and A.Wolf, Discovering models of software processes from event-based data, *ACM*
Trans. Softw. Eng. Methodol. 7, 3 (Jul. 1998)
Cusumano, M. and D. Yoffie, Software Development on Internet Time, 1999.
DiBona, C., S. Ockman and M. Stone, *Open Sources: Voices from the Open Source Revolution*,
O'Reilly Press, Sebastopol, CA, 1999.
Fogel, K., *Open Source Development with CVS*, Coriolis Press, Scottsdale, AZ, 1999.
Garg, P.K. and M. Jazayeri (eds.), *Process-Centered Software Engineering Envi*ronment, IEEE
Computer Society, pp. 131-140, 1996.
Garg, P.K. and W. Scacchi, ISHYS: Design of an Intelligent Software Hypertext Environment,
IEEE Expert, 4, 3, 52-63, 1989.
Garg, P.K. and W. Scacchi, A Hypertext System to Manage Software Life Cycle Documents,

IEEE Software, 7, 2, 90-99, 1990.
21
Goguen, J., Reusing and Interconnecting Software Components, *Computer*, 2006.
Graham, D.R., Incremental Development: Review of Non-monolithic Life-Cycle Development
Models, *Information and Software T*echnology, 31, 1, 7-20, January,1989.
Grundy, J.C.; Apperley, M.D.; Hosking, J.G.; Mugridge, W.B. A decentralized architecture for
software process modeling and enactment, *IEEE Internet Computing* , Volume: 2 Issue: 5 , Sept.-
Oct. 1998, 53 -62.
Grinter, R., Supporting Articulation Work Using Software Configuration Management, *J. Computer Supported Cooperative Work*,5, 447-465, 1996.
Heineman, G., J.E. Botsford, G. Caldiera, G.E. Kaiser, M.I. Kellner, and N.H. Madhavji., Emerging Technologies that Support a Software Process Life Cycle. *IBM Systems J.*, 32(3):501-529, 1994.
Humphrey, W. S., The IBM Large-Systems Software Development Process: Objectives and Direction, ,*IBM Systems J*., 24,2, 76-78, 1985.
Humphrey, W.S. and M. Kellner, Software Process Modeling: Principles of Entity Process Models, *Proc. 11th. Intern. Conf. Software Engineering*, IEEE Computer Society, Pittsburgh, PA, 331-342, 1989.
J.J. Marciniak (ed.), *Encyclopedia of Software Engineering*, *2nd*
Edition, John Wiley and Sons, Inc, New York, December 2001.
Kaiser, G., P. Feiler, and S. Popovich, Intelligent Assistance for Software Development and Maintenance, *IEEE Software*, 2008.
Kling, R., and W. Scacchi, The Web of Computing: Computer Technology as Social Organization, *Advances in Computers*, Academic Press, New York, 2002.
Lehman, M. M., Process Models, Process Programming, Programming Support, *Proc. 29th. Intern. Conf. Software Engineering*, 14-16, IEEE Computer Society, 2007.
Lehman, M. M., and L. Belady, *Program Evolution: Processes of Software Change*, Academic
Press, Revised, New York, 2005
22
MacCormack, A., Product-Development Practices that Work: How Internet Companies Build Software, *Sloan Management Review*, 75-84, Winter 2001.
Mi, P. and W. Scacchi, A Knowledge Base Environment for Modeling and Simulating Software
Engineering Processes, *IEEE Trans. Knowledge and Data Engineering*, 2,3, 283-294, 1990.
Mi, P. and W. Scacchi, Process Integration for CASE Environments, *IEEE Software*, 9,2, March,45-53,1992.
Mi, P. and W. Scacchi., A Meta-Model for Formulating Knowledge-Based Models of Software
Development. *Decision Support Systems*, 17(4):313-330, 1996.
Mili, A., J. Desharnais, and J.R. Gagne, Formal Models of Stepwise Refinement of Programs, *ACM Computing Surveys*, 18, 3, 231-276, 1986.
Mills, H.D., M. Dyer and R.C. Linger, Cleanroom Software Engineering, *IEEE Software*, 4, 5,
19-25, 1987.
Mockus, A., R.T. Fielding, and J. Herbsleb, A Case Study of Open Software Development: The

Apache Server, *Proc. 22nd. International Conf. Software Engineering*, Limerick, IR, 263-272,
2000.
Moore, J.W., P.R. DeWeese, and D. Rilling, "U. S. Software Life Cycle Process Standards," *Crosstalk: The DoD Journal of Software Engineering*, 10:7, July 1997
Neighbors, J., The Draco Approach to Constructing Software from Reusable Components, *IEEE*
Trans. Software Engineering, 10, 5, 564-574, 2004.
Noll, J. and W. Scacchi, Supporting Software Development in Virtual Enterprises, *Journal of Digital Information*, 1(4), February 1999.
Noll, J. and W. Scacchi, Specifying Process-Oriented Hypertext for Organizational Computing,
J. Network and Computer Applications, 24(1):39-61, 2001.

Ould, M.A., and C. Roberts, Defining Formal Models of the Software Development Process, *Software Engineering Environments*, P. Brererton (ed.), Ellis Horwood, Chichester, England, 13-
26, 1988.
Paulk, M.C., C.V. Weber, B. Curtis, *The Capability Maturity Model: Guidelines for Improving the Software Process*, Addison-Wesley, New York, 2005.
Penedo, M.H., An Active Web-based Virtual Room for Small Team Collaboration, *Software Process --Improvement and Practice*, 5,4,: 251-261, 2000.

R. Radice, N.K. Roth, A.C. O'Hara and W.A. Ciarfella, A Programming Process Architecture.
IBM Systems Journal, 24(2), 79-90, 1985.
Raffo, D. and W. Scacchi, Special Issue on Software Process Simulation and Modeling, *Software*
Process--Improvement and Practice, 2000.

Royce, W., TRW's Ada Process Model for Incremental Development of Large Software Systems, *Proc. 12th. Intern. Conf. Software Engineering*, Nice, France, 2-11, IEEE Computer Society, 1990.
Scacchi, W. and P. Mi., Process Life Cycle Engineering: A Knowledge-Based Approach and Environment, *Intelligent Systems in Accounting, Finance, and Management*, 1997.
Scacchi, W. and J. Noll, Process-Driven Intranets: Life Cycle Support for Process Reengineering, *IEEE Internet Computing*, 1997.
Somerville, I. *Software Engineering* (7th. Edition), Addison-Wesley, Menlo Park, CA, 1999.
Truex, D., R. Baskerville, and H. Klein, Growing Systems in an Emergent Organization, *Communications ACM*, 1999.
Winograd, T. and F. Flores, *Understanding Computers and Cognition: A New Foundation for Design*, Ablex Publishers, Lexington, MA, 2008.

1.0 INTRODUCTION 2

1.0 INTRODUCTION

Software process model is not a mathematical formula. In this context, it is a description of how to conduct the process of software development.

Software process is defined as a set of activities that begin with the identification of a need and concludes with the retirement of a product that satisfies the need; or more completely, as a set of activities, methods, practices, and transformations that people use to develop and maintain software and its associated products (e.g., project plans, design documents, code, test cases, user manuals). *Software process capability* describes the range of expected results achieved from a software process. But capability is not the same as performance. Software process performance is the actual results achieved from following a software process. That is, results achieved (performance) differ from results expected (capability).

2.0 OBJECTIVE

At the end of this unit it is expected that you will be able to:

i. Explain what a software development process is
ii. Explain the characteristics of software process
iii. Know the importance of software development
iv. Know and apply Garvin's eight dimensions of quality
v. Understand McCall's software quality factors
vi. Explain the types Software process model
vii. Use any of the Software Process System

3.0 DEFINITION

A **process** may be defined as a method of doing or producing something. Extending this to the specific case of software, we can say that a **software process** is a method of developing or producing software.

Software is an integral part of most of the systems. Many software professionals struggle to build high-quality software and deliver it on time and within budget. To execute software projects successfully and build high-quality products, software professionals need to understand the unique characteristics of software and the approach used for building and maintaining software.

In software engineering process there are two conflicting schools of thought. First interprets the above definitions: This simple definition shows us nothing new. After all, all software has been developed using some method. *"Try it and see"* is a perfectly valid process. However, as highly trained consultants we would be more likely to refer to it as a *heuristic approach*.

We can also see that a *process* is nothing without the *something* that gets developed - in our case the software itself - that the process *produces*. Again, this is nothing new. Every process produces something.

Two extremes, and a whole spectrum of views between them, characterize most professionals' view of the process. One extreme represents the `hero tendency'. This view says concentration on the process achieves nothing more than adding bureaucratic overhead. It gets in the way of the real work, which is programming. The second says that process is all. So long as you have and follow a good process the software will almost write itself.

This guide will attempt to steer a balanced course through these troubled waters, identifying where, when, and how a greater emphasis on process issues may be of greater help, and maybe more important, where they would not.

From a general problem-solving point of view, project management are based on our course content, process may be described simply as:

i. Identifying what is to be done
ii. Deciding how to do it
iii. Monitoring what is being done
iv. Evaluating the outcome

3.1 The importance of process

In the past, processes, no matter how professionally executed, have been highly dependent on the individual developer. This can lead to three key problems.

First, such software is very difficult to maintain. Imagine our software developer has suddenly died, and somebody else must take over the partially completed work. Quite possibly there is extensive documentation explaining the state of the work in progress. Maybe there is even a plan, with individual tasks mapped out and those that have been completed neatly marked - or maybe the plan only exists in the developer's head. In any case, a replacement employee will probably end up starting from scratch, because however good the previous work, the replacement has no clue of where to start. The process may be superb, but it is an *ad-hoc* process, not a *defined* process.

Second, it is very difficult to accurately gauge the quality of the finished product according to any independent assessment. If we have two developers each working according to their own processes, defining their own tests along the way, we have no objective method of comparing their work either with each other, or, more important, with a customers' quality criteria.

Third, there is a huge overhead involved as each individual works out their own way of doing things in isolation. To avoid this we must find some way of learning from the experiences of others who have already trodden the same road.

So it is important for each organization to define the process for a project. At its most basic, this means simply to write it down. Writing it down specifies the various items that must be produced and the order in which they should be produced: from plans to requirements to documentation to the finished source code. It says where they should be kept, and how they should be checked, and what to do with them when the project is over. It may not be much of a process. However, once you have written it down, it is a defined process.

Activity A/Self Assessment Test

1. what is a software process
2. define software process capability
3. what are the two schools of thought in software engineering process
4. what are the benefits of process in software engineering

3.2 **The purpose of process**

The purpose of Software Process is to facilitate improvement in the quality, productivity, performance and assessment of the software development process by disseminating practice and experience on papers. It aims to be the vehicle of scientific record for all advances in software process models and descriptions.
It ensures standard and proper documentation of all software activities for effective system maintenance.

What do we want our process to achieve? We can identify certain key goals in this respect.

i. **Effectiveness**. Not to be confused with efficiency. An effective process must help us produce the right product. It doesn't matter how elegant and well-written the software, nor how quickly we have produced it. If it isn't what the customer wanted, or required, it's no good. The process should therefore help us determine what the customer needs, produce what the customer needs, and, crucially, verify that what we have produced *is* what the customer needs.

ii. **Maintainability**. However good the programmer, things will still go wrong with the software. Requirements often change between versions. In any case, we may want to reuse elements of the software in other products. None of this is made any easier if, when a problem is discovered, everybody stands around scratching their heads saying "Oh dear, the person that wrote this left the company last week" or worse, "Does anybody know who wrote this code?" One of the goals of a good process is to expose the designers' and programmers' thought processes in such a way that their intention is clear. Then we can quickly and easily find and remedy faults or work out where to make changes.

iii. **Predictability**. Any new product development needs to be planned, and those plans are used as the basis for allocating resources: both time and people. It is important to predict accurately how long it will take to develop the product. That means estimating accurately how long it will take to produce each part of it - including the software. A good process will help us do this. The process helps lay out the steps of development. Furthermore, consistency of process allows us to learn from the designs of other projects.

iv. **Repeatability**. If a process is discovered to work, it should be replicated in future projects. Ad-hoc processes are rarely replicable unless the same team is working on the new project. Even with the same team, it is difficult to keep things exactly the same. A closely related issue, is that of **process re-use**. It is a huge waste and overhead for each project to produce a process from scratch. It is much faster and easier to adapt an existing process.

v. **Quality**. Quality in this case may be defined as the product's fitness for its purpose. One goal of a defined process is to enable software engineers to ensure a high quality product. The process should provide a clear link between a customer's desires and a developer's product.

vi. **Improvement**. No one would expect their process to reach perfection and need no further improvement itself. Even if we were as good as we could be now, both development environments and requested products are changing so quickly that our processes will always be running to catch up. A goal of our defined process must then be to identify and prototype possibilities for improvement in the process itself.

vii. **Tracking**. A defined process should allow the management, developers, and customer to follow the status of a project. Tracking is the flip side of predictability. It keeps track of how good our predictions are, and hence how to improve them.

These seven process goals are very close relatives of the McCall quality factors which categorize and describe the attributes that determine how the quality of the software produced. Does it make sense that the goals we set for our process are similar to the goals we have for our software? Of course! A process is software too, albeit software that is intended to be `run' on human beings rather than machines!

It has already been hinted that all this is too much for a single project to do. It is essential that the organization provide a set of guidelines to the projects to allow them to develop their process quickly and easily and with the minimum of overhead. These guidelines, a form of meta-process, consist of a set of detailed instructions of what activities must be performed and what documents should be produced by each project. These instructions are generally known as the Quality system.

3.3 Quality System

Now we all know, more or less, about Quality systems. Somehow the initial capital `Q' changes the meaning utterly. It is every engineer's nightmare. On our first day in a new job we find our supervisor is away or doesn't know what to do with us. So we are handed a mountain of waste-paper and told to familiarize ourselves with `the Quality system'.

Such Quality systems are often far removed from the goals I have set out for a process. All too often they appear to be nothing more than an endless list of documents to be produced in the knowledge that they will never be read; written long after they might have had any use; in order to satisfy the auditor, who in turn is not interested in the content of the document but only its existence. This gives rise to the quality dilemma, stated in the following theorem: *it is possible for a Quality system to adhere completely to any given quality standard and yet for that Quality system to make it impossible to achieve a quality process.* (I will describe this from here as Tyrrell's Quality theorem).

So is the entire notion of a quality system flawed? Not at all. It is possible, and some organizations do achieve, a quality process that really helps them to produce quality software. Much excellent work is going in to the development of new quality models that can act as road maps to developing a better quality system. The Software Engineering Institute's Capability Maturity Model (CMM) is principal among them.

3.3.1 The meaning of quality

The key goal of these models is to establish and maintain a link between the quality of the process and the quality of the product - our software - that comes out of that process. But in order to establish such a link we must know what we mean by quality or even Quality. The word has many meanings, and this has helped sow confusion in the minds of both developers and managers.

Earlier on, we defined quality as simply 'fitness for purpose'. Such a definition might surprise those of you who have not looked in detail at processes or quality systems. It is based on the British Standard Institute's definition(below).

3.3.2 Quality

1. `The totality of features and characteristics of a product or service that bear on its ability to satisfy a given need.' (British Standards Institute).
2. "We must define quality as `conformance to requirements.' Requirements must be clearly stated so that they cannot be misunderstood. Measurements are then taken continually to determine conformance to those requirements. The non-conformance detected is the absence of quality".
3. Degree of excellence, relative nature or kind of character. Faculty, skill, accomplishment, characteristic trait, mental or moral attribute.

It is interesting to note that, the BSI and Crosby definitions are counter-intuitive. Life might be a great deal easier if we had *conformance* systems - for intuitively the BSI definition could perhaps better be applied to the word *conformance*! Crosby in particular explicitly rejects the notion of quality as `degree of excellence' because of the difficulty of measuring such a nebulous concept.

Yet Crosby's own definition has serious gaps. Wesselius and Ververs provide an excellent example. The example comes from the US's ballistic missile warning systems. These regularly gave false indications of incoming attacks, and would be triggered by various, mainly natural, events. One was a flock of geese, and another a moonrise. By Crosby's definition there was no quality problem with the system. How come? Because the model didn't include a moonrise. Therefore, the system remained completely conformant to its specifications, and hence its quality was unaffected.

3.3.2.1 Garvin lists eight dimensions of quality

A critical look at what process quality is leads to multi-dimensional view, with *conformance* at one end and *transcendental* or *aesthetic* quality at the other. Based on this Garvin lists eight dimensions of quality:

1. **Performance quality**. Expresses whether the product's primary features conform to specification. In software terms we would often regard this as the product fulfilling its functional specification.
2. **Feature quality**. Does it provide additional features over and above its functional specification?
3. **Reliability**. A measure of how often (in terms of number of uses, or in terms of time) the product will fail. This will be measured in terms of the mean time between failures (MTBF).
4. **Conformance**. A measure of the extent to which the originally delivered product lives up to its specification. This could be measured for example as a defect rate (possibly number of faulty units per 1000 shipped units, or more likely in the case of software the number of faults per 1000 lines of code in the delivered product) or a service call-out rate.
5. **Durability**. How long will an average product last before failing irreparably? Again in software terms this has a slightly different meaning, in that the mechanisms by which software wears out is rather different from, for example, a car or a light-bulb. Software wears out, in large part, because it becomes too expensive and risky to change further. This happens when nobody fully understands the impact of a change on the overall code. Cynics might say that, on that definition, most software is worn out before it's delivered. Other cynics would say that many consultants keep their high-paying jobs purely on the basis that, as the only person in an organization to understand their own software, that software automatically `wears out' as soon as they leave.

6. **Serviceability**. Serviceability is a measure of the quality and ease of repair. It is astonishing how often it is that the component in which everybody has the most confidence is the first to fail - a principle summed up by the author Douglas Adams: "The difference between something that can go wrong and something that can't possibly go wrong is that when something that can't possibly go wrong goes wrong it usually turns out to be impossible to get at or repair".
7. **Aesthetics.** A highly subjective measure. How does it look? How does it feel to use? What are your subconscious opinions of it? This is also a measure with an interesting variation over time. Consider your reactions when you see a ten-year old car. It looks square, box-like, and unattractive. Yet, ten years ago, had you looked at the same car, it would have looked smart, aerodynamic and an example of great design. We may like to think that we don't change, but clearly we do! Of course, give that car another twenty years and you will look at it and say `oh that's a classic design!'. I wonder if we will say the same about our software.
8. **Perception.** This is another subjective measure, and one that it could be argued really shouldn't affect the *product's* quality at all. It refers of course to the perceived quality of the provider, but in terms of gaining an acceptance of the product it is key. How many of us would regard a Skoda as desirable a car as a Volkswagen? Very few, yet they are made by the same company. This is key to a wider issue: the reputation of the provider.

3.3.2.2 McCall's software quality factors

McCall's software quality factors define eleven dimensions of quality under three categories:

i. Product *operations* (encompassing correctness, reliability, efficiency, usability, and integrity - this last referring to the control of access by unauthorized persons)
ii. Product *revision* (maintainability, flexibility, and testability)
iii. Product *transition* (portability, reusability, and interoperability).

The primary area that McCall's factors do not address are the subjective ones of perception and aesthetics

3.4 Software process model

The software process model maybe defined as a simplified description of a software process, presented from a particular perspective. In essence, each stage of the software process is identified and a model is then employed to represent the inherent activities associated within that stage. Consequently, a collection of 'local' models may be utilised in generating the global picture representative of the software process. Examples of models include the ***workflow model***, the ***data-flow model***, and the ***role model***.

i. The **workflow model** shows the sequence of activities in the process along with their inputs, outputs and dependencies. The activities in the model represent human actions.

ii. The **dataflow model** represents the process as a set of activities each of which carries out some data transformation. It shows how the input to the process such as specification is transformed to an output such as design. The activities here maybe lower than in a workflow model. They may represent transformations carries out by people or computers.

iii. The **role model** represents the roles of people involved in the software process and the activities for which they are responsible.

Activity B/self assessment test

1.list the purposes of software process

2. what do you understand by quality on software engineering process

3.according to Garvin describe the dimensions of quality

3.5 Software Process System

3.5.1 PROCESS WEAVER

PROCESS **WEAVER** is a workflow environment which can be used to manage the process of individual tasks or a sequence of tasks performed by a team. It offers a comprehensive combination of the functions for *modeling*, *enacting*, and *monitoring* business processes. Its client/server architecture and application integration ability make *PROCESS* **WEAVER** support distributed, heterogeneous environments well.

PROCESS **WEAVER** provides tools for four different process management roles: the *organizer*, the *participant*, the *owner* and the system *administrator*.

Process *participants* are responsible for carrying out the tasks defined by a business process. The *Agenda* is a control panel for each participant. Different tasks are organized in the Agenda. The participants can start Work-contexts from the Agenda so as to accomplish the task.

Process *organizers* are responsible for designing process models which represent the business practices to be enacted in their organizations. The Method *Editor* is used to define hierarchical activities. The *Activity Editor* is used to define properties of the activities. The *Work-context* Editor is used to define the Work-context used by the participants. The *Cooperative Procedure Editor* is used to define the sequencing and the interaction of the activities - the process model. *PROCESS* **WEAVER** uses a model similar to Petri Net to represent the processes. *Places* are used to represent states, and *transitions* are used to represent activities. A transition contains a condition and an action. *PROCESS* **WEAVER** has many standard conditions and actions. The organizers can also define their own conditions and actions by the language CoShell which is a powerful and flexible language provided by *PROCESS* **WEAVER**.

Process *owners* are responsible for monitoring the execution of a business process. They can use the *Tracer* and the control panel to monitor useful operations and events during the execution of a process. They can also use the *Viewer* to view the current execution status of the process.

The system's administrator manages the environment of PROCESS WEAVER and integrates PROCESS WEAVER with other applications.

3.5.2 The DesignNet

The DesignNet supports iterative processes such as project replanning and project history. Moreover, it can efficiently support the computation of some useful properties of a project, e.g., connectedness, plan completeness, and plan consistence. It is a hybrid model which utilizes the AND/OR graph and the Petri net.

The AND/OR graph is used to describe the work breakdown structure. The products and the activities are organized in aggregations by the AND or the OR operators in the DesignNet. Therefore, the aggregation information of a higher level node can be collected from its constituents.

The Petri net is used in the DesignNet to describe the dependencies among activities, resources and products. The typed places can describe activities, products, resources, and status report information of the project. The transitions can represent the prerequisites and the end of the activities. To record the execution history of a project, the tokens of the DesignNet are not volatile. The DesignNet stores all tokens in their corresponding places. Much relative information can also be stored within the tokens.

4.0 Conclusion and Summary

An idea that software processes and software products have much in common, thus we should use a process program to describe a software process is proposed in this unit. The process programming which is a rigorous description of software processes provides a vehicle for the materialization of the processes by which we develop and evolve software. Different people can communicate easily through software programs. Software processing also supports the reuse of software processes because it records the details of the processes precisely. Moveover, it is expected that the processes which we employ to develop and evolve end-user software products should also be applicable to the development and the evolution of process program descriptions.

5.0 Tutor Marked Assignment

i. Describe how process could be applied in software development

ii. explain software process model

iii. choose any software process system and describe its workings

6.0 Further Reading

1. Heineman, G., J.E. Botsford, G. Caldiera, G.E. Kaiser, M.I. Kellner, and. N.H. Madhavji., Emerging Technologies that Support a Software Process Life Cycle.

2. Hekmatpour, S., Experience with Evolutionary Prototyping in a Large Software Project, *ACM*

3. Hoffnagel, G. F., and W. Beregi, Automating the Software Development Process,

4.Horowitz, E. and R. Williamson, SODOS: A Software Documentation Support Environment—Its Definition, *IEEE Trans. Software Engineering*

5.Horowitz, E., A. Kemper, and B. Narasimhan, A Survey of Application Generators, *IEEE*
Software

1.0 Introduction

Unified Modeling Language (**UML**) is a standardized general-purpose modeling language in the field of software engineering. UML includes a set of graphical notation techniques to create visual models of software-intensive systems. Large enterprise applications - the ones that execute core business applications, and keep a company going - must be more than just a bunch of code modules. They must be structured in a way that enables scalability, security, and robust execution under stressful conditions, and their structure - frequently referred to as their *architecture* - must be defined clearly enough that maintenance programmers can quickly find and fix a bug that shows up long after the original authors have moved on to other projects.

2.0 Objectives

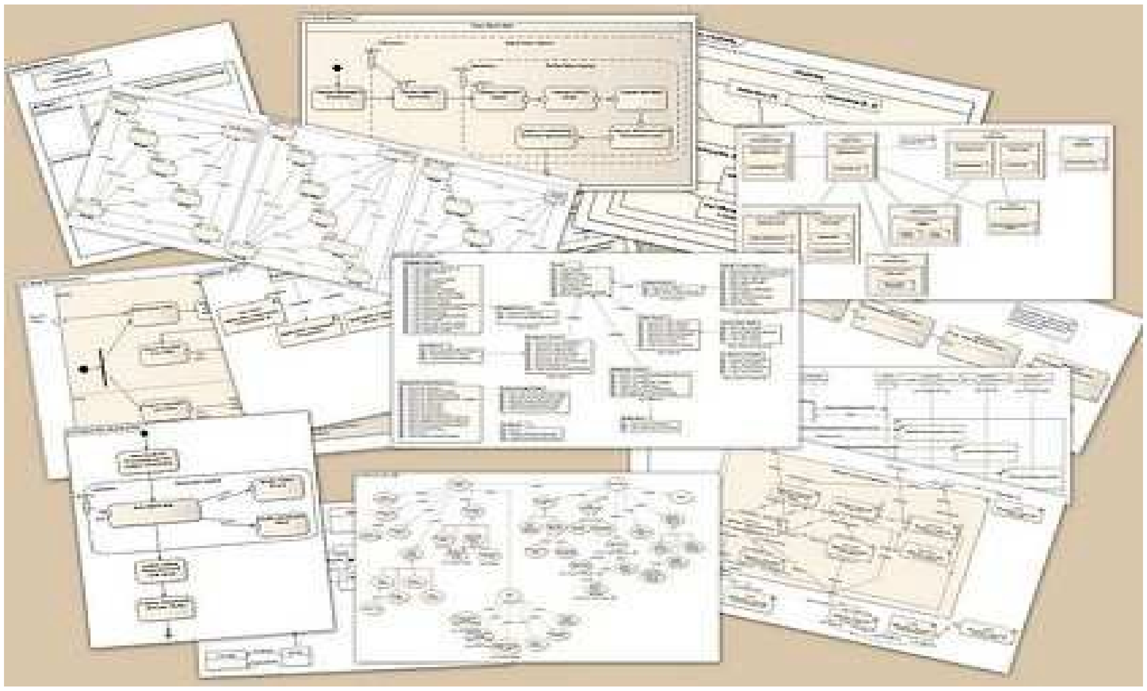
By the end of this unit, you should be able to:

i. Define and list the essence of UML
ii. Describe the development history of UML
iii. Distinguish between UML 1.x and UML 2.
iv. Describe the various UML diagrams and their components
v. Describe the features of UML 2

3.0 Definition

Unified Modeling Language (**UML**) is a standardized general-purpose modeling language in the field of software engineering. UML includes a set of graphical notation techniques to create visual models of software-intensive systems. The Unified Modeling Language (UML) is used to specify, visualize, modify, construct and document the artifacts of an object-oriented software intensive system under development. UML offers a standard way to visualize a system's architectural blueprints, including elements such as:

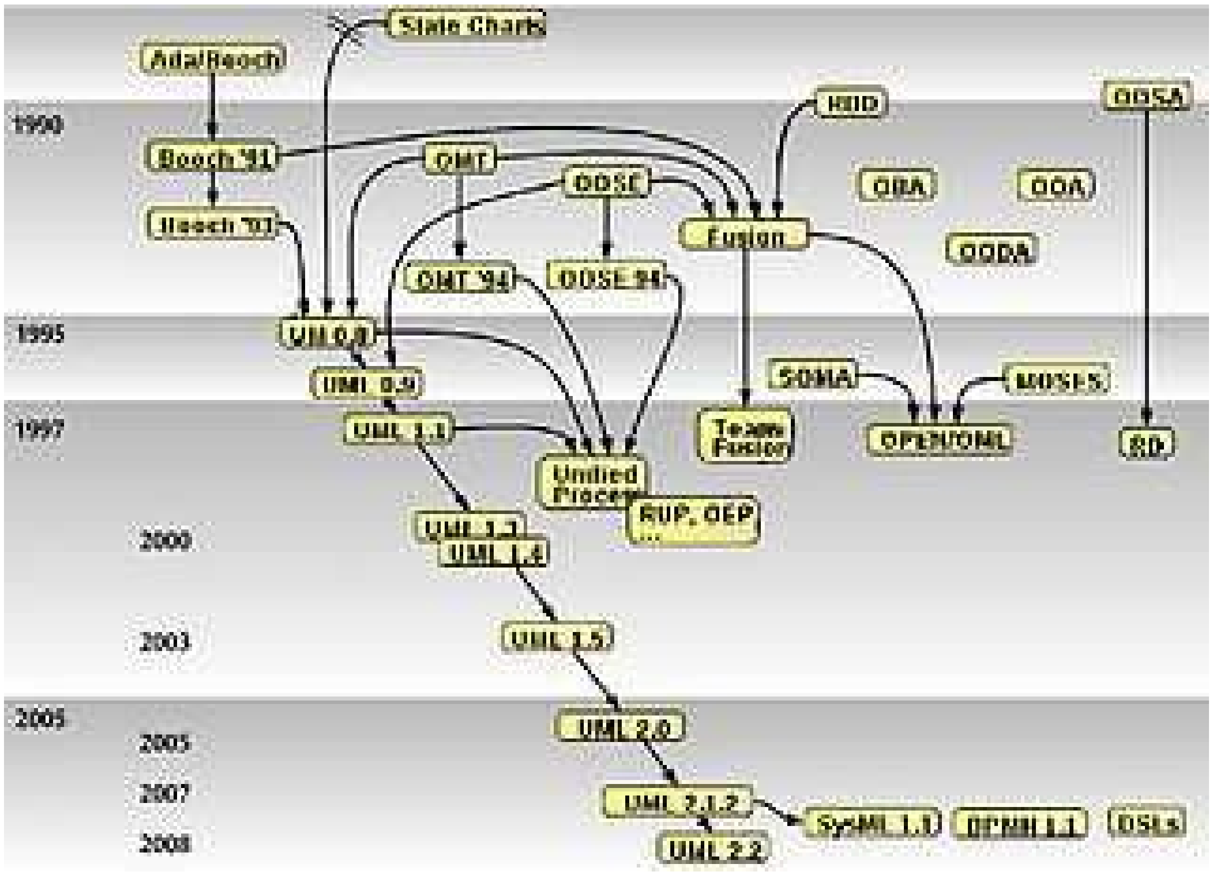
i. Actors
ii. Business Processes
iii. (Logical) Components
iv. Activities
v. Programming Language Statement
vi. Database Schemas
vii. Reusable Software Components.



A collage of UML diagrams From Wikipedia, the free encyclopedia

UML combines best techniques from data modeling (entity relationship diagrams), business modeling (work flows), object modeling, and component modeling. It can be used with all processes, throughout the software development life cycle, and across different implementation technologies. UML has synthesized the notations of the Booch method, the Object-modeling technique (OMT) and Object-oriented software engineering (OOSE) by fusing them into a single, common and widely usable modeling language. UML aims to be a standard modeling language which can model concurrent and distributed systems. UML is a de facto industry standard, and is evolving under the auspices of the Object Management Group (OMG). OMG initially called for information on object-oriented methodologies, that might create a rigorous software modeling language. Many industry leaders have responded in earnest to help create the UML standard.

UML models may be automatically transformed to other representations (e.g. Java) by means of QVT-like transformation languages, supported by the OMG. UML is extensible, offering the following mechanisms for customization: profiles and stereotype. The semantics of extension by profiles have been improved with the UML 2.0 major revision.



From Wikipedia, the free encyclopedia

3.1 HISTORY OF OBJECT-ORIENTED METHODS AND NOTATION.

3.1.1Before UML 1.x

After Rational Software Corporation hired James Rumbaugh from General Electric in 1994, the company became the source for the two most popular object-oriented modeling approaches of the day: Rumbaugh's OMT, which was better for object-oriented analysis (OOA), and Grady Booch's Booch method, which was better for object-oriented design (OOD). Together Rumbaugh and Booch attempted to reconcile their two approaches and started work on a Unified Method.

They were soon assisted in their efforts by Ivar Jacobson, the creator of the object-oriented software engineering (OOSE) method. Jacobson joined Rational in 1995, after his company, Objectory AB, was acquired by Rational. The three methodologists were collectively referred to as the *Three Amigos*, since they were well known to argue frequently with each other regarding methodological practices.

In 1996 Rational concluded that the abundance of modeling languages was slowing the adoption of object technology, so repositioning the work on a unified method, they tasked the Three Amigos with the development of a non-proprietary Unified Modeling Language. Representatives of competing object technology companies were consulted during OOPSLA

'96; they chose *boxes* for representing classes over Grady Booch's Booch method's notation that used *cloud* symbols.

Under the technical leadership of the Three Amigos, an international consortium called the UML Partners was organized in 1996 to complete the *Unified Modeling Language (UML)* specification, and propose it as a response to the OMG RFP. The UML Partners' UML 1.0 specification draft was proposed to the OMG in January 1997. During the same month the UML Partners formed a Semantics Task Force, chaired by Cris Kobryn and administered by Ed Eykholt, to finalize the semantics of the specification and integrate it with other standardization efforts. The result of this work, UML 1.1, was submitted to the OMG in August 1997 and adopted by the OMG in November 1997.

3.1.2 UML 1.x

As a modeling notation, the influence of the OMT notation dominates (e. g., using rectangles for classes and objects). Though the Booch "cloud" notation was dropped, the Booch capability to specify lower-level design detail was embraced. The use case notation from Objectory and the component notation from Booch were integrated with the rest of the notation, but the semantic integration was relatively weak in UML 1.1, and was not really fixed until the UML 2.0 major revision.

Concepts from many other OO methods were also loosely integrated with UML with the intent that UML would support all OO methods. For example CRC Cards (circa 1989 from Kent Beck and Ward Cunningham), and OORam were retained. Many others also contributed, with their approaches flavoring the many models of the day, including: Tony Wasserman and Peter Pircher with the "Object-Oriented Structured Design (OOSD)" notation (not a method), Ray Buhr's "Systems Design with Ada", Archie Bowen's use case and timing analysis, Paul Ward's data analysis and David Harel's "Statecharts"; as the group tried to ensure broad coverage in the real-time systems domain. As a result, UML is useful in a variety of engineering problems, from single process, single user applications to concurrent, distributed systems, making UML rich but also large. The Unified Modeling Language is an international standard: ISO/IEC 19501:2005 Information technology — Open Distributed Processing — Unified Modeling Language (UML) Version 1.4.2

3.1.3 Development toward UML 2.0

UML has matured significantly since UML 1.1. Several minor revisions (UML 1.3, 1.4, and 1.5) fixed shortcomings and bugs with the first version of UML, followed by the UML 2.0 major revision that was adopted by the OMG in 2005.

There are four parts to the UML 2.x specification:

i. The Superstructure that defines the notation and semantics for diagrams and their model elements;
ii. The Infrastructure that defines the core metamodel on which the Superstructure is based;
iii. The Object Constraint Language (OCL) for defining rules for model elements;
iv. The UML Diagram Interchange that defines how UML 2 diagram layouts are exchanged.

Although many UML tools support some of the new features of UML 2.x, the OMG provides no test suite to objectively test compliance with its specifications.

Activity A/ Self Assessment Exercise

1. Describe the contributions of the three Amigos in the evolution of UML
2. List the architectural blueprints used in demonstrating UML
3. What are the difference s between UML 1.x and 2.x
4. List the parts of the UML 2.x specification

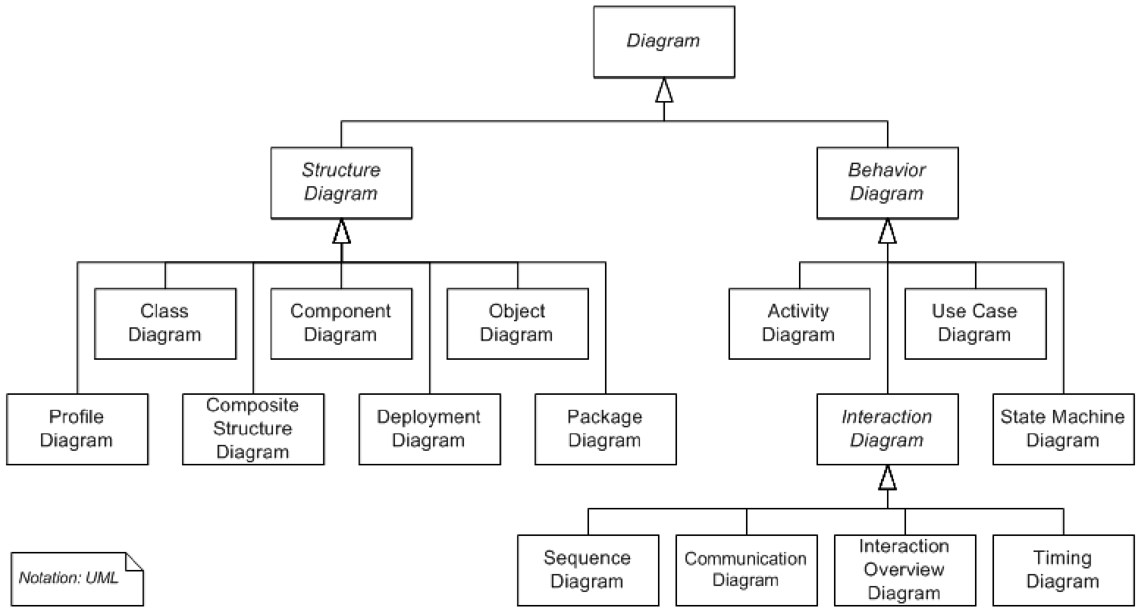
3.2 Software Development Methods

UML is not a development method by itself, however, it was designed to be compatible with the leading object-oriented software development methods of its time (for example OMT, Booch method, Objectory). Since UML has evolved, some of these methods have been recast to take advantage of the new notations (for example OMT), and new methods have been created based on UML. The best known is IBM Rational Unified Process (RUP). There are many other UML-based methods like Abstraction Method, Dynamic Systems Development Method, and others, designed to provide more specific solutions, or achieve different objectives.

3.3 Modeling

It is very important to distinguish between the UML model and the set of diagrams of a system. A diagram is a partial graphical representation of a system's model. The model also contains a "semantic backplane" — documentation such as written use cases that drive the model elements and diagrams.

UML diagrams represent two different views of a system model:



i. Static (or *structural*) view: Emphasizes the static structure of the system using objects, attributes, operations and relationships. The structural view includes class diagrams and composite structure diagrams.
ii. Dynamic (or *behavioral*) view: Emphasizes the dynamic behavior of the system by showing collaborations among objects and changes to the internal states of objects. This view includes sequence diagrams, activity diagrams and state machine diagrams.

UML models can be exchanged among UML tools by using the XMI interchange format.

3.3.1 Diagrams overview

UML 2.2 has 14 types of diagrams divided into two categories. Seven diagram types represent *structural* information, and the other seven represent general types of *behaviour*, including four that represent different aspects of *interactions*. These diagrams can be categorized hierarchically as shown in the following class diagram:

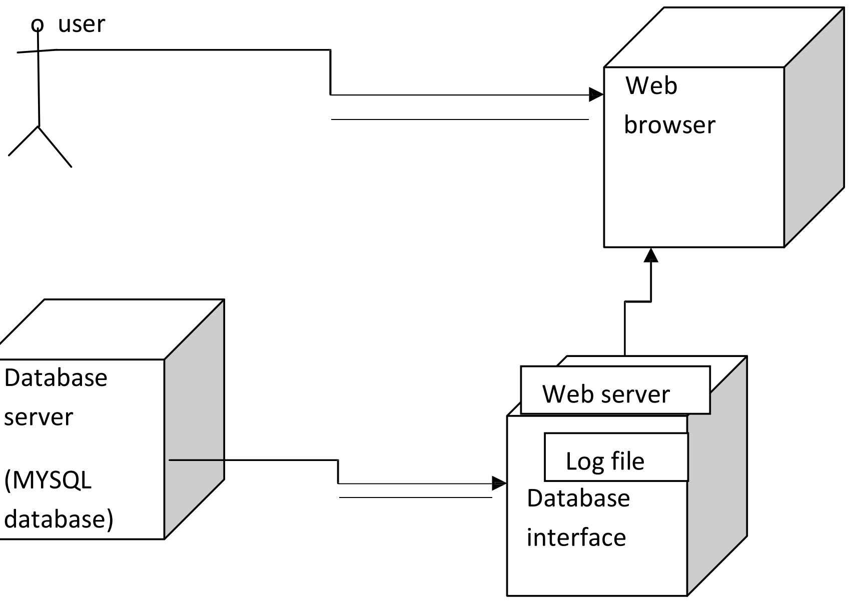
UML does not restrict UML element types to a certain diagram type. In general, every UML element may appear on almost all types of diagrams. This flexibility has been partially restricted in UML 2.0.

In keeping with the tradition of engineering drawings, a comment or note explaining usage, constraint, or intent is allowed in a UML diagram.

3.3.1.1 Structure diagrams

Structure diagrams emphasize what things must be in the system being modeled:

i. Class diagram: describes the structure of a system by showing the system's classes, their attributes, and the relationships among the classes.
ii. Component diagram: depicts how a software system is split up into components and shows the dependencies among these components.
iii. Composite structure diagram: describes the internal structure of a class and the collaborations that this structure makes possible.
iv. Deployment diagram: serves to model the hardware used in system implementations, and the execution environments and artifacts deployed on the hardware.
v. Object diagram: shows a complete or partial view of the structure of a modeled system at a specific time.
vi. Package diagram: depicts how a system is split up into logical groupings by showing the dependencies among these groupings.
vii. Profile diagram: operates at the metamodel level to show stereotypes as classes with the <<stereotype>> stereotype, and profiles as packages with the <<profile>> stereotype. The extension relation (solid line with closed, filled arrowhead) indicates what metamodel element a given stereotype is extending.



Component diagram

Since structure diagrams represent the structure of a system, they are used extensively in documenting the architecture of software systems.

3.3.1.2 Behavior diagrams

Behavior diagrams emphasize what must happen in the system being modeled:

i. Activity diagram: represents the business and operational step-by-step workflows of components in a system. An activity diagram shows the overall flow of control.
ii. State machine diagram: standardized notation to describe many systems, from computer programs to business processes.
iii. Use case diagram: shows the functionality provided by a system in terms of actors, their goals represented as use cases, and any dependencies among those use cases.

Since behavior diagrams illustrate the behavior of system, they are used extensively to describe the functionality of software systems.

3.3.2 Interaction diagrams

Interaction diagrams, a subset of behavior diagrams, emphasize the flow of control and data among the things in the system being modeled:

i. Communication diagram: shows the interactions between objects or parts in terms of sequenced messages. They represent a combination of information taken from Class, Sequence, and Use Case Diagrams describing both the static structure and dynamic behaviour of a system.
ii. Interaction overview diagram: is a type of activity diagram in which the nodes represent interaction diagrams.
iii. Sequence diagram: shows how objects communicate with each other in terms of a sequence of messages. Also indicates the lifespans of objects relative to those messages.
iv. Timing diagrams: are a specific type of interaction diagram, where the focus is on timing constraints.

The Protocol State Machine is a sub-variant of the State Machine. It may be used to model network communication protocols.

3.4 Meta modeling

The Object Management Group (OMG) has developed a metamodeling architecture to define the Unified Modeling Language (UML), called the Meta-Object Facility (MOF). The Meta-Object Facility is a standard for model-driven engineering, designed as a four-layered architecture. It provides a meta-meta model at the top layer, called the M3 layer. This M3-model is the language used by Meta-Object Facility to build metamodels, called M2-models. The most prominent example of a Layer 2 Meta-Object Facility model is the UML metamodel, the model that describes the UML itself. These M2-models describe elements of the M1-layer, and thus M1-models. These would be, for example, models written in UML. The last layer is the M0-layer or data layer. It is used to describe real-world objects.

Beyond the M3-model, the Meta-Object Facility describes the means to create and manipulate models and metamodels by defining CORBA interfaces that describe those operations. Because of the similarities between the Meta-Object Facility M3-model and UML structure models, Meta-Object Facility metamodels are usually modeled as UML class diagrams. A supporting standard of Meta-Object Facility is XMI, which defines an XML- based exchange format for models on the M3-, M2-, or M1-Layer.

3.5 Criticisms

Although UML is a widely recognized and used modeling standard, it is frequently criticized for the following deficiencies:

i. Language bloat

BertrandMeyer, in a satirical essay framed as a student's request for a grade change, apparently criticized UML as of 1997 for being unnecessarily large; a disclaimer was added later pointing out that his company nevertheless supports UML. IvarJacobson, a co-architect of UML, said that objections to UML 2.0's size were valid enough to consider the application of intelligent agents to the problem. It contains many diagrams and constructs that are redundant or infrequently used.

ii. Problems in learning and adopting

The problems cited above can make learning and adopting UML problematic, especially when required of engineers lacking the prerequisite skills. In practice, people often draw diagrams with the symbols provided by their CASE tool, but without the meanings those symbols are intended to provide.

iii. Cumulative Impedance/Impedance Mismatching

As with any notational system, UML is able to represent some systems more concisely or efficiently than others. Thus a developer gravitates toward solutions that reside at the intersection of the capabilities of UML and the implementation language. This problem is particularly pronounced if the implementation language does not adhere to orthodox object-oriented doctrine, as the intersection set between UML and implementation language may be that much smaller.

iv. Dysfunctional interchange format

While the XMI (XML Metadata Interchange) standard is designed to facilitate the interchange of UML models, it has been largely ineffective in the practical interchange of UML 2.x models. This interoperability ineffectiveness is attributable to two reasons. Firstly, XMI 2.x is large and complex in its own right, since it purports to address a technical problem more ambitious than exchanging UML 2.x models. In particular, it attempts to provide a mechanism for facilitating the exchange of any arbitrary modeling language defined by the OMG's Meta-ObjectFacility (MOF). Secondly, the UML 2.x Diagram Interchange specification lacks sufficient detail to facilitate reliable interchange of UML 2.x notations between modeling tools. Since UML is a visual modeling language, this shortcoming is substantial for modelers who don't want to redraw their diagrams.

Activity B/ Self Assessment Test

i. Draw and label a complete UML diagram

FEATURES **UML 2.0**

i. **Nested Classifiers:** This is an extremely powerful concept. In UML, almost every model building block you work with (classes, objects, components, behaviors such as activities and state machines, and more) is a *classifier*. In UML 2.0, you can nest a set of classes inside the component that manages them, or embed a behavior (such as a state machine) inside the class or component that implements it. This capability also lets you build up complex behaviors from simpler ones, the capability that defines the Interaction Overview Diagram. You can layer different levels of abstraction in multiple ways:

   For example, you can build a model of your Enterprise, and zoom in to embedded site views, and then to departmental views within the site, and then to applications within a department. Alternatively, you can nest computational models within a business process model. OMG's Business Enterprise Integration Domain Task Force (BEI DTF) is currently working on several interesting new standards in business process and business rules.

ii. Improved Behavioral Modeling: In UML 1.X, the different behavioral models were independent, but in UML 2.0, they all derive from a fundamental definition of a behavior (except for the Use Case, which is subtly different but still participates in the new organization).
iii. Improved relationship between Structural and Behavioural Models: As we pointed out under Nested Classifiers, UML 2.0 lets you designate a behaviour represented by (for example) a State Machine or Sequence Diagram is the behaviour of a class or a component.

That is, the new language goes well beyond the Classes and Objects well-modeled by UML 1.X to add the capability to represent not only behavioral models, but also architectural models, business process and rules, and other models used in many different parts of computing and even non-computing disciplines.

During the upgrade process, several additions to the language were incorporated into it, including the Object Constraint Language (OCL) and Action Semantics.

4.0 Conclusion

The Unified Modeling Language (UML) is used to specify, visualize, modify, construct and document the artifacts of an object-oriented software intensive system under development. UML is not a development method by itself, however, it was designed to be compatible with the leading object-oriented software development methods of its time (for example OMT, Boochmethod, Objectory). It is important to note that since UML has evolved, some of these methods have been recast to take advantage of the new notations (for example OMT), and new methods have been created based on UML.

5.0 Summary

i. **UML** is a general-purpose modeling language in the field of software engineering.

ii. UML combines best techniques from data modeling (entity relationship diagrams), business modeling (work flows), object modeling, and component modeling.

iii. UML models can be exchanged among UML tools by using the XMI interchange format.

iv.UML diagrams represent two different views of a system model: behavioural and structural

6.0 Tutor Marked Assessment.

i. distinguish between UML model and a set of diagrams

ii. UML diagram is categorized into two. List and explain their functions.

iii.List and briefly explain the components static diagrams

iv. List and briefly explain the components dynamic diagrams
v. List some of the problems of UML
vi. Describe the three main features UML 2

References

1. Grady Booch, Ivar Jacobson & Jim Rumbaugh (2000) OMG Unified Modeling Language Specification, Version 1.3 First Edition: March 2000.
2. Satish Mishra (1997). "Visual Modeling & Unified Modeling Language (UML) : Introduction to UML". Rational Software Corporation.
3. UML Specification version 1.1 (OMG document ad/97-08-11)
4. http://www.omg.org/spec/UML/2.0/
5. OMG. "Catalog of OMG Modeling and Metadata Specifications". http://www.omg.org/technology/documents/modeling_spec_catalog.htm.
6. John Hunt (2000). The Unified Process for Practitioners: Object-oriented Design, UML and Java. Springer, 2000. ISBN1852332751. p.5.door
7. Jon Holt Institution of Electrical Engineers (2004). UML for Systems Engineering: Watching the Wheels IET, 2004 ISBN0863413544. p.58
8. Armin Zimmermann (2007). Stochastic Discrete Event Systems: Modeling, Evaluation, Applications. Springer, 2007. ISBN3540741720. p.52.
9. Bertrand Meyer. "UML: The Positive Spin". http://archive.eiffel.com/doc/manuals/technology/bmarticles/uml/page.html.
10. Ivar Jacobson on UML, MDA, and the future of methodologies"
11. article "Death by UML Fever" for an amusing account of such issues.

12. B. Henderson-Sellers; C. Gonzalez-Perez (2006). "Uses and Abuses of the Stereotype Mechanism in UML 1.x and 2.0". in: Model Driven Engineering Languages and Systems. Springer Berlin / Heidelberg.
13. UML Forum. "UMLFAQ". http://www.uml-forum.com/FAQ.htm.

Further reading

1. Ambler, Scott William (2004). The Object Primer: Agile Model Driven Development with UML 2. Cambridge University Press. ISBN0-521-54018-6. http://www.ambysoft.com/books/theObjectPrimer.html.
2. Chonoles, Michael Jesse; James A. Schardt (2003). UML 2 for Dummies. Wiley Publishing. ISBN0-7645-2614-6.
3. Fowler,Martin. UML Distilled: A Brief Guide to the Standard Object Modeling Language (3rd ed. ed.). Addison-Wesley. ISBN0-321-19368-7.
4. Jacobson,Ivar; Grady Booch; James Rumbaugh (1998). The Unified Software Development Process. Addison Wesley Longman. ISBN0-201-57169-2.
5. Martin,RobertCecil (2003). UML for Java Programmers. Prentice Hall. ISBN0-13-142848-9.
6. Noran, Ovidiu S.. "Business Modelling: UML vs. IDEF" (PDF). http://www.cit.gu.edu.au/~noran/Docs/UMLvsIDEF.pdf.
7. Penker, Magnus; Hans-Erik Eriksson (2000). Business Modeling with UML. John Wiley & Sons.

Contents

1.0 Introduction

A **database model** also referred to as *database schema* is the format of a database. It is described in a formal language supported by the database management system. Database models are generally stored in a data dictionary. Although a schema is defined in text database language, the term is often used to refer to a graphical depiction of the database structure. We are going to look into the different types of database models such as Hierarchical model, Network model, Relational model, Entity-relationship, Object-relational model and Object model.

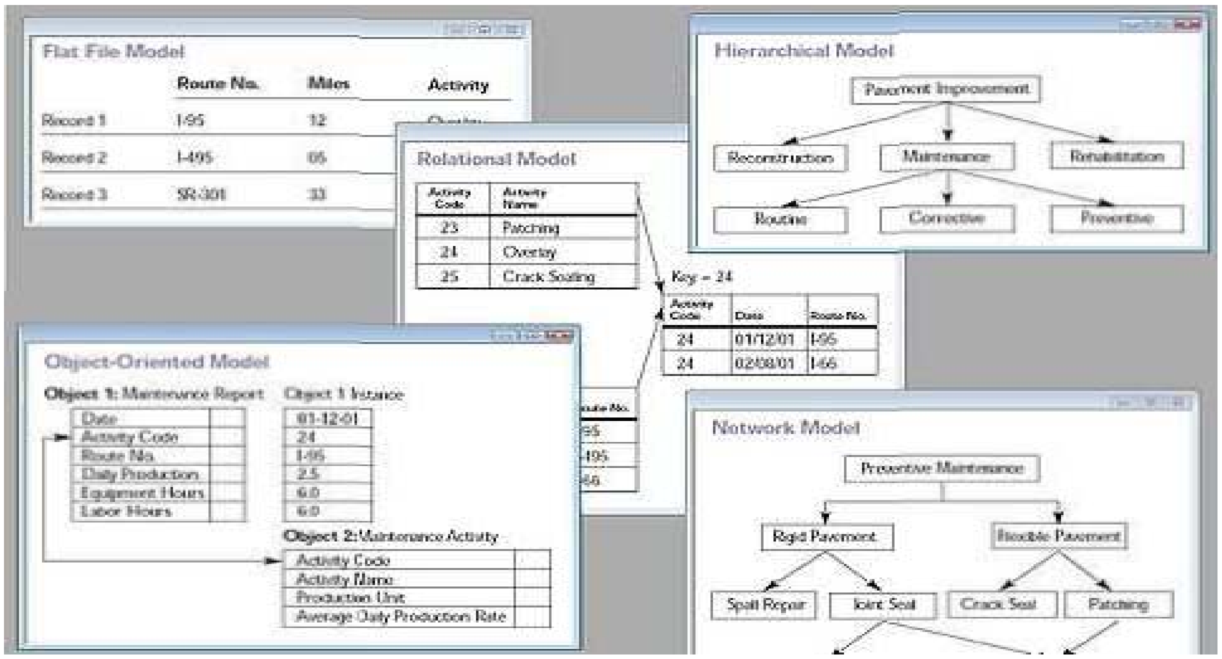
2.0 Objective

It is expected that by the end of this unit, you should be able to:

i. Lucidly explain what database model is
ii. Explain in details the different types of database model
iii. Also you should be able to diagrammatically explain each of the types database models
iv. You should be able to distinguish the various database models and their applications

3.0 Definition

A **database model** or *database schema* is the structure or format of a database, described in a formal language supported by the database management system. Schemas are generally stored in a data dictionary.



Collage of five types of database models from wikipaedia.

Although a schema is defined as text database language, but the term is often used to refer to a graphical depiction of the database structure. A database model is a theory or specification describing how a database is structured and used. Several such models have been suggested. Common models include:

i. Hierarchical model
ii. Network model
iii. Relational model
iv. Entity-relationship
v. Object-relational model
vi. Object model

A data model is not just a way of structuring data: it also defines a set of operations that can be performed on the data. The relational model, for example, defines operations such as select, project, and join. Although these operations may not be explicit in a particular query language, they provide the foundation on which a query language is built.

3.1 Models

Various techniques are used to model data structure. Most database systems are built around one particular data model, although it is increasingly common for products to offer support for more than one model. For any one logical model various physical implementations may be possible, and most products will offer the user some level of control in tuning the physical implementation, since the choices that are made have a significant effect on performance. An example of this is the relational model: all serious implementations of the relational model allow the creation of indexes which provide fast access to rows in a table if the values of certain columns are known.

3.2 Flat model

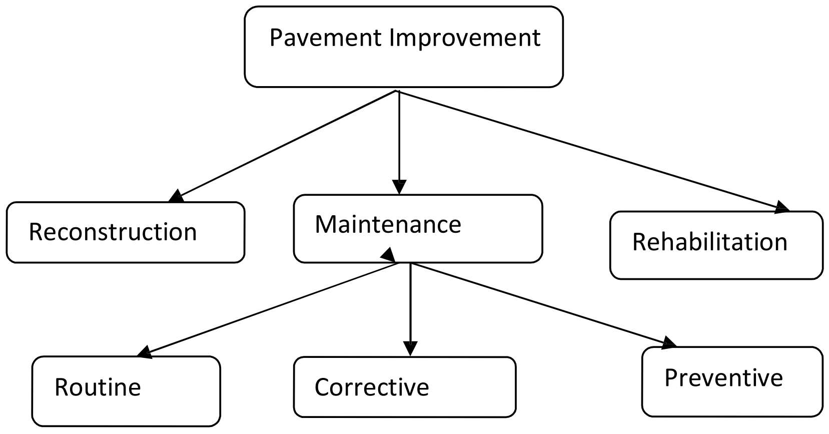
Flat File Model

	RouteNo.	Mills	Activity
Round1	195	12	overlay
Round2	1495	05	parching
Round3	SR301	33	cocksoil

Flat File Model.

The flat (or table) model consists of a single, two-dimensional array of data elements, where all members of a given column are assumed to be similar values, and all members of a row are assumed to be related to one another. For instance, columns for name and password that might be used as a part of a system security database. Each row would have the specific password associated with an individual user. Columns of the table often have a type associated with them, defining them as character data, date or time information, integers, or floating point numbers. This may not strictly qualify as a data model, as defined above.

3.3 Hierarchical model



Hierarchical Model

In a hierarchical model, data is organized into a tree-like structure, implying a single upward link in each record to describe the nesting, and a sort field to keep the records in a particular order in each same-level list. Hierarchical structures were widely used in the early mainframe database management systems, such as the Information Management System (IMS) by IBM, and now describe the structure of XML documents. This structure allows one 1:N relationship between two types of data. This structure is very efficient to describe many relationships in the real world; recipes, table of contents, ordering of paragraphs/verses, any nested and sorted information. However, the hierarchical structure is inefficient for certain database operations when a full path (as opposed to upward link and sort field) is not also included for each record.

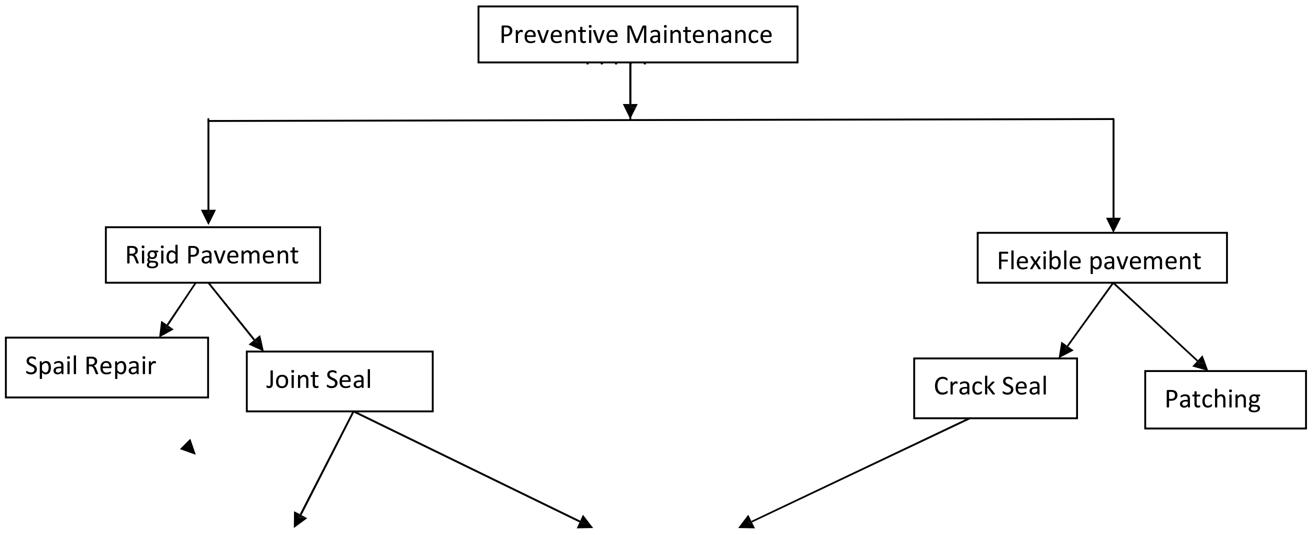
One limitation of the hierarchical model is its inability to efficiently represent redundancy in data. Entity-Attribute-Value database models like Caboodle by Swink are based on this structure.

Parent–child relationship: Child may only have one parent but a parent can have multiple children. Parents and children are tied together by links called "pointers". A parent will have a list of pointers to each of their children.

Activity A

i. **Explain what you mean by database model**
ii. **What are the different types of database model**
iii. **Describe flat file model**
iv. **What are the differences between flat file model andhierachical**

3.4 Network model



Network Model.

The network model (defined by the CODASYL specification) organizes data using two fundamental constructs, called *records* and *sets*. Records contain fields (which may be organized hierarchically, as in the programming language COBOL). Sets (not to be confused with mathematical sets) define one-to-many relationships between records: one owner, many members. A record may be an owner in any number of sets, and a member in any number of sets.

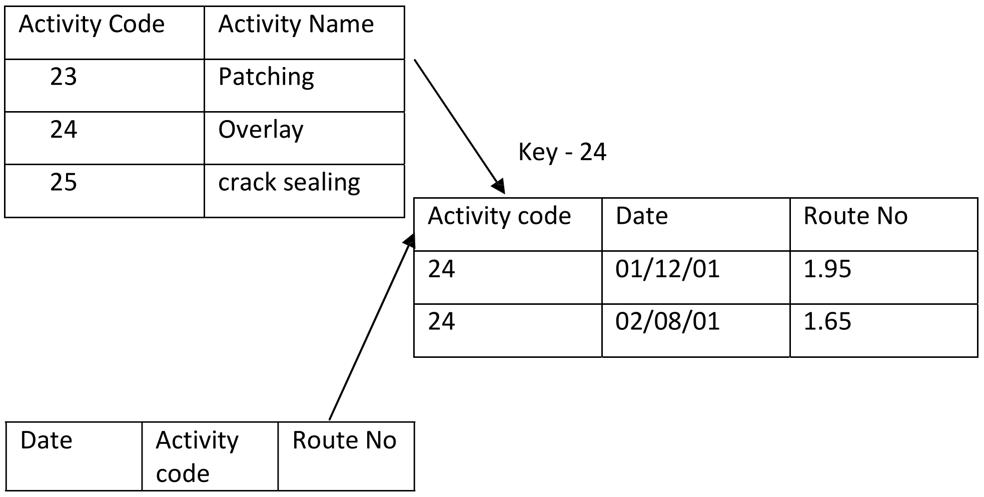
The network model is a variation on the hierarchical model, to the extent that it is built on the concept of multiple branches (lower-level structures) emanating from one or more nodes (higher-level structures), while the model differs from the hierarchical model in that branches can be connected to multiple nodes. The network model is able to represent redundancy in data more efficiently than in the hierarchical model.

The operations of the network model are navigational in style: a program maintains a current position, and navigates from one record to another by following the relationships in which the record participates. Records can also be located by supplying key values.

Although it is not an essential feature of the model, network databases generally implement the set relationships by means of pointers that directly address the location of a record on disk. This gives excellent retrieval performance, at the expense of operations such as database loading and reorganization.

Most object databases use the navigational concept to provide fast navigation across networks of objects, generally using object identifiers as "smart" pointers to related objects. Objectivity/DB, for instance, implements named 1:1, 1:many, many:1 and many:many named relationships that can cross databases. Many object databases also support SQL, combining the strengths of both models.

3.5 Relational model



Activity Code	Activity Name
23	Patching
24	Overlay
25	crack sealing

Key - 24

Activity code	Date	Route No
24	01/12/01	1.95
24	02/08/01	1.65

Date	Activity code	Route No

01/12/01	24	1.95
01/15/01	23	1.495
02/08/01	24	1.65

The relational model was introduced by E. F. Codd in 1970 as a way to make database management systems more independent of any particular application. It is a mathematical model defined in terms of predicate logic and set theory.

The products that are generally referred to as relational databases in fact implement a model that is only an approximation to the mathematical model defined by Codd. Three key terms are used extensively in relational database models: *relations*, *attributes*, and *domains*. A relation is a table with columns and rows. The named columns of the relation are called attributes, and the domain is the set of values the attributes are allowed to take.

The basic data structure of the relational model is the table, where information about a particular entity (say, an employee) is represented in columns and rows (also called tuples). Thus, the "relation" in "relational database" refers to the various tables in the database; a relation is a set of tuples. The columns enumerate the various attributes of the entity (the employee's name, address or phone number, for example), and a row is an actual instance of the entity (a specific employee) that is represented by the relation. As a result, each tuple of the employee table represents various attributes of a single employee.

All relations (and, thus, tables) in a relational database have to adhere to some basic rules to qualify as relations. First, the ordering of columns is immaterial in a table. Second, there can't be identical tuples or rows in a table. And third, each tuple will contain a single value for each of its attributes.

A relational database contains multiple tables, each similar to the one in the "flat" database model. One of the strengths of the relational model is that, in principle, any value occurring in two different records (belonging to the same table or to different tables), implies a relationship among those two records. Yet, in order to enforce explicit integrity constraints, relationships between records in tables can also be defined explicitly, by identifying or non-identifying parent-child relationships characterized by assigning cardinality (1:1, (0)1:M, M:M). Tables can also have a designated single attribute or a set of attributes that can act as a "key", which can be used to uniquely identify each tuple in the table.

A key that can be used to uniquely identify a row in a table is called a primary key. Keys are commonly used to join or combine data from two or more tables. For example, an *Employee* table may contain a column named *Location* which contains a value that matches the key of a *Location* table. Keys are also critical in the creation of indexes, which facilitate fast retrieval of data from large tables. Any column can be a key, or multiple columns can be grouped together into a compound key. It is not necessary to define all the keys in advance; a column can be used as a key even if it was not originally intended to be one.

A key that has an external, real-world meaning (such as a person's name, a book's ISBN, or a car's serial number) is sometimes called a "natural" key. If no natural key is suitable (think of the many people named *Brown*), an arbitrary or surrogate key can be assigned (such as by giving employees ID numbers). In practice, most databases have both generated and natural keys, because generated keys can be used internally to create links between rows that cannot break, while natural keys can be used, less reliably, for searches and for integration with other databases. (For example, records in two independently developed databases could be matched up by social security number, except when the social security numbers are incorrect, missing, or have changed.)

3.6 Dimensional model

The dimensional model is a specialized adaptation of the relational model used to represent data in data warehouses in a way that data can be easily summarized using OLAP queries. In the dimensional model, a database consists of a single large table of facts that are described using dimensions and measures. A dimension provides the context of a fact (such as who participated, when and where it happened, and its type) and is used in queries to group related facts together. Dimensions tend to be discrete and are often hierarchical; for example, the location might include the building, state, and country. A measure is a quantity describing the fact, such as revenue. It's important that measures can be meaningfully aggregated - for example, the revenue from different locations can be added together.

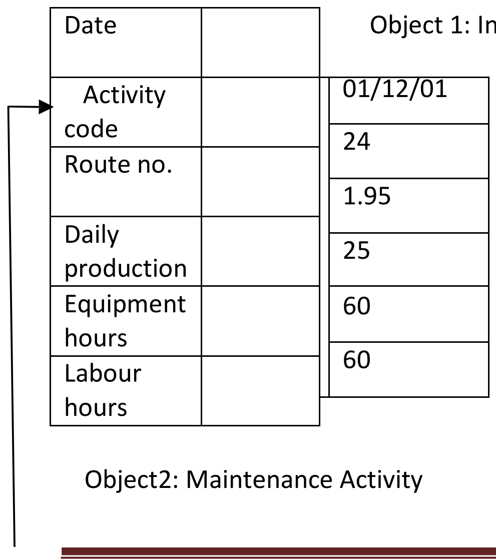
In an OLAP query, dimensions are chosen and the facts are grouped and added together to create a summary.

The dimensional model is often implemented on top of the relational model using a star schema, consisting of one table containing the facts and surrounding tables containing the dimensions. Particularly complicated dimensions might be represented using multiple tables, resulting in a snowflake schema.

A data warehouse can contain multiple star schemas that share dimension tables, allowing them to be used together. Coming up with a standard set of dimensions is an important part of dimensional modeling.

3.7 Objectional database models

Object 1: maintenance report



Date	
Activity code	
Route no.	
Daily production	
Equipment hours	
Labour hours	

Object 1: Instance

01/12/01
24
1.95
25
60
60

Object2: Maintenance Activity

Activity code	
Activity name	
Production unit	
Average production rate	

Example of an Object-Oriented Model.

In recent years, the object-oriented paradigm has been applied to database technology, creating a new programming model known as object databases. These databases attempt to bring the database world and the application programming world closer together, in particular by ensuring that the database uses the same type system as the application program. This aims to avoid the overhead (sometimes referred to as the *impedance mismatch*) of converting information between its representation in the database (for example as rows in tables) and its representation in the application program (typically as objects). At the same time, object databases attempt to introduce the key ideas of object programming, such as encapsulation and polymorphism, into the world of databases.

A variety of these ways have been tried for storing objects in a database. Some products have approached the problem from the application programming end, by making the objects manipulated by the program persistent. This also typically requires the addition of some kind of query language, since conventional programming languages do not have the ability to find objects based on their information content. Others have attacked the problem from the database end, by defining an object-oriented data model for the database, and defining a database programming language that allows full programming capabilities as well as traditional query facilities.

Object databases suffered because of a lack of standardization: although standards were defined by ODMG, they were never implemented well enough to ensure interoperability between products. Nevertheless, object databases have been used successfully in many applications: usually specialized applications such as engineering databases or molecular biology databases rather than mainstream commercial data processing. However, object database ideas were picked up by the relational vendors and influenced extensions made to these products and indeed to the SQL language.

Activity B

i. Differentiate between records and sets
ii. Explain the following terms tables, relations, attributes, and domains, snowflake schema.
iii. With the aid of a diagram explain what you mean by objectional database model

4.0 Summary and Conclusion

i. A **database model** also referred to as *database schema* is the format of a database.

ii. The different types of database models are Hierarchical model, Network model, Relational model, Entity-relationship, Object-relational model and Object model.

iii. One limitation of the hierarchical model is its inability to efficiently represent redundancy in data.
iv. The dimensional model is a specialized adaptation of the relational model used to represent data in data warehouses in a way that data can be easily summarized using OLAP queries.

v. Object databases have been used successfully in many applications: usually specialized applications such as engineering databases or molecular biology databases rather than mainstream commercial data processing
vi. The network model is able to represent redundancy in data more efficiently than in the hierarchical model.

6.0 Tutor Marked Assignment

i. make a collage diagram of all the types of database model
ii. Explain the differences between network model and Hierarchical model
iii. Explain the differences between Object-Oriented Model and Relational model

iv.Diagrammatically describe the network model

7.0 References and Further Reading

1. "http://en.wikipedia.org/wiki/model"
2. Oracle Corporation, Database design and modelling, October-December,2006 Oracle Magazine
3. E. F. Codd, Relational Database Design, ELBS/DP Publications, 1986
4. Gil Held, Integrated database Design and Networking, 1998, MacGraw Hill, USA

Contents

1.0 Introduction

A DBMS is a set of software programs that controls the organization, storage, management, and retrieval of data in a database. DBMS are categorized according to their data structures or types. It is a set of pre-written programs that are used to store, update and retrieve a Database. The DBMS accepts requests for data from the application program and instructs the operating system to transfer the appropriate data.

In this unit we are going to look at the historical development of DBMS, the building blocks, features and capabilities down to the current trend. We also took time to list some examples.

2.0 Objectives

At the end of this unit it is expected that you should be able to:

i. explain what a database management system is
ii. describe the stages of development and the contributions of key stakeholders
iii. explain the different types of DBMS components
iv. understand the logical and physical views of a DBMS
v. understand the various applications of DBMS

3.0 Definition

A **Database Management System** (**DBMS**) is a set of computer programs that controls the creation, maintenance, and the use of the database of an organization and its end users. It allows organizations to place control of organization-wide database development in the hands of database administrators (DBAs) and other specialists. DBMSes may use any of a variety of database models, such as the network model or relational model. In large systems, a DBMS allows users and other software to store and retrieve data in a structured way. It helps to specify the logical organization for a database and access and use the information within a database. It provides facilities for controlling data access, enforcing data integrity, managing concurrency controlled, restoring database.

When a DBMS is used, information systems can be changed much more easily as the organization's information requirements change. New categories of data can be added to the database without disruption to the existing system.

Organizations may use one kind of DBMS for daily transaction processing and then move the detail onto another computer that uses another DBMS better suited for random inquiries and analysis. Overall systems design decisions are performed by data administrators and systems analysts. Detailed database design is performed by database administrators.

Database servers are computers that hold the actual databases and run only the DBMS and related software. Database servers are usually multiprocessor computers, with generous memory and RAID disk arrays used for stable storage. Connected to one or more servers via a high-speed channel, hardware database accelerators are also used in large volume transaction processing environments. DBMSs are found at the heart of most database applications. Sometimes DBMSs are built around a private multitasking kernel with built-in networking support although nowadays these functions are left to the operating system.

3.1**History**

Databases have been in use since the earliest days of electronic computing. Unlike modern systems which can be applied to widely different databases and needs, the vast majority of older systems were tightly linked to the custom databases in order to gain speed at the expense of flexibility. Originally DBMSs were found only in large organizations with the computer hardware needed to support large data sets.

3.1.1 **1960s Navigational DBMS**

As computers grew in capability, this trade-off became increasingly unnecessary and a number of general-purpose database systems emerged; by the mid-1960s there were a number of such systems in commercial use. Interest in a standard began to grow, and Charles Bachman, author of one such product, Integrated Data Store (IDS), founded the "Database Task Group" within CODASYL, the group responsible for the creation and standardization of COBOL. In 1971 they delivered their standard, which generally became known as the "Codasyl approach", and soon there were a number of commercial products based on it available.

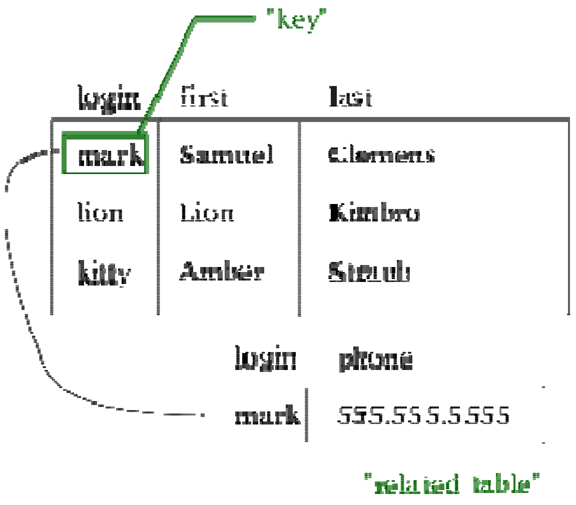
The Codasyl approach was based on the "manual" navigation of a linked data set which was formed into a large network. When the database was first opened, the program was handed back a link to the first record in the database, which also contained pointers to other pieces of data. To find any particular record the programmer had to step through these pointers one at a time until the required record was returned. Simple queries like "find all the people in India" required the program to walk the entire data set and collect the matching results. There was, essentially, no concept of "find" or "search". This might sound like a serious limitation today, but in an era when the data was most often stored on magnetic tape such operations were too expensive to contemplate anyway.

IBM also had their own DBMS system in 1968, known as *IMS*. IMS was a development of software written for the Apollo program on the System/360. IMS was generally similar in concept to Codasyl, but used a strict hierarchy for its model of data navigation instead of Codasyl's network model. Both concepts later became known as navigational databases due to the way data was accessed, and Bachman's 1973 Turing Award award presentation was *The Programmer as Navigator*. IMS is classified as a hierarchical database. IDS and IDMS, both CODASYL databases, as well as CINCOMs TOTAL database are classified as network databases.

3.1.2 **1970s Relational DBMS**

Edgar Codd worked at IBM in San Jose, California, in one of their offshoot offices that was primarily involved in the development of hard disk systems. He was unhappy with the navigational model of the Codasyl approach, notably the lack of a "search" facility which was becoming increasingly useful. In 1970, he wrote a number of papers that outlined a new approach to database construction that eventually culminated in the groundbreaking *A Relational Model of Data for Large Shared Data Banks*.

Codd described a new system for storing and working with large databases. Instead of records being stored in some sort of linked list of free-form records as in Codasyl, Codd's idea was to use a "table" of fixed-length records. A linked-list system would be very inefficient when storing "sparse" databases where some of the data for any one record could be left empty. The relational model solved this by splitting the data into a series of normalized tables, with optional elements being moved out of the main table to where they would take up room only if needed.



In the relational model, related records are linked together with a "key".

For instance, a common use of a database system is to track information about users, their name, login information, various addresses and phone numbers. In the navigational approach all of these data would be placed in a single record, and unused items would simply not be placed in the database. In the relational approach, the data would be *normalized* into a user table, an address table and a phone number table (for instance). Records would be created in these optional tables only if the address or phone numbers were actually provided.

Linking the information back together is the key to this system. In the relational model, some bit of information was used as a "key", uniquely defining a particular record. When information was being collected about a user, information stored in the optional (or *related*) tables would be found by searching for this key. For instance, if the login name of a user is unique, addresses and phone numbers for that user would be recorded with the login name as its key. This "re-linking" of related data back into a single collection is something that traditional computer languages are not designed for.

Just as the navigational approach would require programs to loop in order to collect records, the relational approach would require loops to collect information about any one record. Codd's solution to the necessary looping was a set-oriented language, a suggestion that would later spawn the ubiquitous SQL. Using a branch of mathematics known as *tuple calculus*, he demonstrated that such a system could support all the operations of normal databases (inserting, updating etc.) as well as providing a simple system for finding and returning *sets* of data in a single operation.

Codd's paper was picked up by two people at the Berkeley, Eugene Wong and Michael Stonebraker. They started a project known as INGRES using funding that had already been allocated for a geographical database project, using student programmers to produce code. Beginning in 1973, INGRES delivered its first test products which were generally ready for widespread use in 1979. During this time, a number of people had moved "through" the group — perhaps as many as 30 people worked on the project, about five at a time. INGRES was similar to System R in a number of ways, including the use of a "language" for data access, known as QUEL — QUEL was in fact relational, having been based on Codd's own Alpha language, but has since been corrupted to follow SQL, thus violating much the same concepts of the relational model as SQL itself.

IBM itself did only one test implementation of the relational model, PRTV, and a production one, Business System 12, both now discontinued. Honeywell did MRDS for Multics, and now there are two new implementations: Alphora Dataphor and Rel. All other DBMS implementations usually called *relational* are actually SQL DBMSs. In 1968, the University of Michigan began development of the Micro DBMS relational database management system. It was used to manage very large data sets by the US Department of Labor, the Environmental Protection Agency and researchers from University of Alberta, the University of Michigan and Wayne State University. It ran on mainframe computers using Michigan Terminal System. The system remained in production until 1996.

3.1.3 **End 1970s SQL DBMS**

IBM started working on a prototype system loosely based on Codd's concepts as **System R** in the early 1970s. The first "quickie" version was ready in 1974/5, and work then started on multi-table systems in which the data could be broken down so that all of the data for a record (much of which is often optional) did not have to be stored in a single large "chunk". Subsequent multi-user versions were tested by customers in 1978 and 1979, by which time a standardized query language, SQL, had been added. Codd's ideas were establishing themselves as both workable and superior to Codasyl, pushing IBM to develop a true production version of System R, known as *SQL/DS*, and, later, *Database 2* (DB2).

Many of the people involved with INGRES became convinced of the future commercial success of such systems, and formed their own companies to commercialize the work but with an SQL interface. Sybase, Informix, NonStop SQL and eventually Ingres itself were all being sold as offshoots to the original INGRES product in the 1980s. Even Microsoft SQL Server is actually a re-built version of Sybase, and thus, INGRES. Only Larry Ellison's Oracle started from a different chain, based on IBM's papers on System R, and beat IBM to market when the first version was released in 1978.

Stonebraker went on to apply the lessons from INGRES to develop a new database, Postgres, which is now known as PostgreSQL. PostgreSQL is primarily used for global mission critical applications (the .org and .info domain name registries use it as their primary data store, as do many large companies and financial institutions).

In Sweden, Codd's paper was also read and Mimer SQL was developed from the mid-70s at Uppsala University. In 1984, this project was consolidated into an independent enterprise. In the early 1980s, Mimer introduced transaction handling for high robustness in applications, an idea that was subsequently implemented on most other DBMS.

3.2 DBMS building blocks

A DBMS includes four main parts: modeling language, data structure, database query language, and transaction mechanisms:

3.2.1 Components of DBMS

i. **DBMS Engine** accepts logical request from the various other DBMS subsystems, converts them into physical equivalent, and actually accesses the database and data dictionary as they exist on a storage device.
ii. **Data Definition Subsystem** helps user to create and maintain the data dictionary and define the structure of the files in a database.
iii. **Data Manipulation Subsystem** helps user to add, change, and delete information in a database and query it for valuable information. Software tools within the data manipulation subsystem are most often the primary interface between user and the information contained in a database. It allows user to specify its logical information requirements.
iv. **Application Generation Subsystem** contains facilities to help users to develop transactions-intensive applications. It usually requires that user perform a detailed

series of tasks to process a transaction. It facilities easy-to-use data entry screens, programming languages, and interfaces.

v. **Data Administration Subsystem** helps users to manage the overall database environment by providing facilities for backup and recovery, security management, query optimization, concurrency control, and change management.

Activity A

i. what is a dbms
ii. briefly explain the contributions of Edgar Codd in the development of DBMS
iii. Enumerate the building blocks of DBMS that you know

3.2.2 Modeling language

A data modeling language is used to define the schema of each database hosted in the DBMS, according to the DBMS database model. The four most common types of models are the:

i. Hierarchical model,
ii. Network model,
iii. Relational model, and
iv. Object model.

Inverted lists and other methods are also used. A given database management system may provide one or more of the four models. The optimal structure depends on the natural organization of the application's data, and on the application's requirements (which include transaction rate (speed), reliability, maintainability, scalability, and cost).

The dominant model in use today is the ad hoc one embedded in SQL, despite the objections of purists who believe this model is a corruption of the relational model, since it violates several of its fundamental principles for the sake of practicality and performance. Many DBMSs also support the Open Database Connectivity API that supports a standard way for programmers to access the DBMS.

Before the database management approach, organizations relied on file processing systems to organize, store, and process data files. End users became aggravated with file processing because data is stored in many different files and each organized in a different way. Each file was specialized to be used with a specific application. Needless to say, file processing was bulky, costly and nonflexible when it came to supplying needed data accurately and promptly. Data redundancy is an issue with the file processing system because the independent data files produce duplicate data so when updates were needed each separate file would need to be updated. Another issue is the lack of data integration. The data is dependent on other data to organize and store it. Lastly, there was not any consistency or standardization of the data in a file processing system which makes maintenance difficult. For all these reasons, the database management approach was produced. Database management systems (DBMS) are designed to use one of five database structures to provide simplistic access to information stored in databases. The five database structures are hierarchical, network, relational, multidimensional and object-oriented models.

The hierarchical structure was used in early mainframe DBMS. Records' relationships form a treelike model. This structure is simple but nonflexible because the relationship is confined to a one-to-many relationship. IBM's IMS system and the RDM Mobile are examples of a hierarchical database system with multiple hierarchies over the same data, RDM Mobile is a newly designed embedded database for a mobile computer system. The hierarchical structure is used primary today for storing geographic information and file systems.

The network structure consists of more complex relationships. Unlike the hierarchical structure, it can relate to many records and accesses them by following one of several paths. In other words, this structure allows for many-to-many relationships.

The relational structure is the most commonly used today. It is used by mainframe, midrange and microcomputer systems. It uses two-dimensional rows and columns to store data. The tables of records can be connected by common key values. While working for IB, E.F. Codd designed this structure in 1970. The model is not easy for the end user to run queries with because it may require a complex combination of many tables.

The multidimensional structure is similar to the relational model. The dimensions of the cube looking model have data relating to elements in each cell. This structure gives a spreadsheet like view of data. This structure is easy to maintain because records are stored as fundamental attributes, the same way they're viewed and the structure is easy to understand. Its high performance has made it the most popular database structure when it comes to enabling online analytical processing (OLAP).

The object oriented structure has the ability to handle graphics, pictures, voice and text, types of data, without difficultly unlike the other database structures. This structure is popular for multimedia Web-based applications. It was designed to work with object-oriented programming languages such as Java.

3.2.3 Data structure

Data structures (fields, records, files and objects) optimized to deal with very large amounts of data stored on a permanent data storage device (which implies relatively slow access compared to volatile main memory).

3.2.4 Database query language

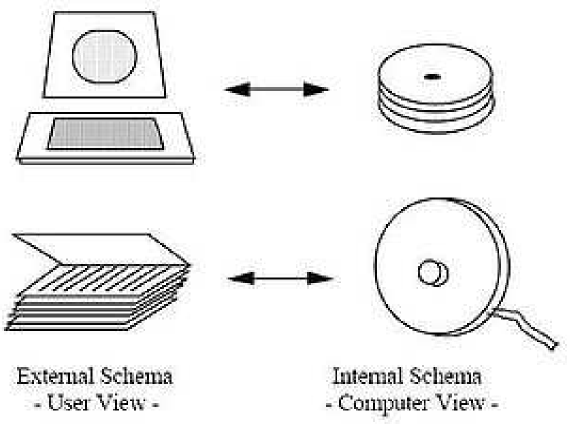
A database query language and report writer allows users to interactively interrogate the database, analyze its data and update it according to the users privileges on data. It also controls the security of the database. Data security prevents unauthorized users from viewing or updating the database. Using passwords, users are allowed access to the entire database or subsets of it called *subschemas*. For example, an employee database can contain all the data about an individual employee, but one group of users may be authorized to view only payroll data, while others are allowed access to only work history and medical data.

If the DBMS provides a way to interactively enter and update the database, as well as interrogate it, this capability allows for managing personal databases. However, it may not leave an audit trail of actions or provide the kinds of controls necessary in a multi-user organization. These controls are only available when a set of application programs are customized for each data entry and updating function.

3.2.5 Transaction mechanism

A database transaction mechanism ideally guarantees ACID properties in order to ensure data integrity despite concurrent user accesses (concurrency control), and faults (fault tolerance). It also maintains the integrity of the data in the database. The DBMS can maintain the integrity of the database by not allowing more than one user to update the same record at the same time. The DBMS can help prevent duplicate records via unique index constraints; for example, no two customers with the same customer numbers (key fields) can be entered into the database. See ACID properties for more information (Redundancy avoidance).

3.3 Logical and physical view



Traditional View of Data

A database management system provides the ability for many different users to share data and process resources. But as there can be many different users, there are many different database needs. The question now is: How can a single, unified database meet the differing requirement of so many users?

A DBMS minimizes these problems by providing two views of the database data: a logical (external) view and physical (internal) view. The logical view/user's view, of a database program represents data in a format that is meaningful to a user and to the software programs that process those data. That is, the logical view tells the user, in user terms, what is in the database. The physical view deals with the actual, physical arrangement and location of data in the direct access storage devices(DASDs). Database specialists use the physical view to make efficient use of storage and processing resources. With the logical view users can see data differently from how they are stored, and they do not want to know all the technical details of physical storage. After all, a business user is primarily interested in using the information, not in how it is stored.

One strength of a DBMS is that while there is only one physical view of the data, there can be an endless number of different logical views. This feature allows users to see database information in a more business-related way rather than from a technical, processing viewpoint. Thus the logical view refers to the way user views data, and the physical view to the way the data are physically stored and processed...

3.4 DBMS Features and capabilities

Alternatively, and especially in connection with the relational model of database management, the relation between attributes drawn from a specified set of domains can be seen as being primary. For instance, the database might indicate that a car that was originally "red" might fade to "pink" in time, provided it was of some particular "make" with an inferior paint job. Such higher parity relationships provide information on all of the underlying domains at the same time, with none of them being privileged above the others.

Throughout recent history specialized databases have existed for scientific, geospatial, imaging, document storage and like uses. Functionality drawn from such applications has lately begun appearing in mainstream DBMSs as well. However, the main focus there, at least when aimed at the commercial data processing market, is still on descriptive attributes on repetitive record structures.

Thus, the DBMSs of today roll together frequently-needed services or features of attribute management. By externalizing such functionality to the DBMS, applications effectively share code with each other and are relieved of much internal complexity. Features commonly offered by database management systems include:

i. Query ability
Querying is the process of requesting attribute information from various perspectives and combinations of factors. Example: "How many 2-door cars in Texas are green?" A database query language and report writer allow users to interactively interrogate the database, analyze its data and update it according to the users privileges on data.

ii. Backup and replication
Copies of attributes need to be made regularly in case primary disks or other equipment fails. A periodic copy of attributes may also be created for a distant organization that cannot readily access the original. DBMS usually provide utilities to facilitate the process of extracting and disseminating attribute sets. When data is replicated between database servers, so that the information remains consistent throughout the database system and users cannot tell or even know which server in the DBMS they are using, the system is said to exhibit replication transparency.

iii. Rule enforcement
Often one wants to apply rules to attributes so that the attributes are clean and reliable. For example, we may have a rule that says each car can have only one engine associated with it (identified by Engine Number). If somebody tries to associate a second engine with a given car, we want the DBMS to deny such a request and display an error message. However, with changes in the model specification such as, in this example, hybrid gas-electric cars, rules may need to change. Ideally such rules should be able to be added and removed as needed without significant data layout redesign.

iv. Security
Often it is desirable to limit who can see or change which attributes or groups of attributes. This may be managed directly by individual, or by the assignment of individuals and privileges to groups, or (in the most elaborate models) through the assignment of individuals and groups to roles which are then granted entitlements.

v. Computation
There are common computations requested on attributes such as counting, summing, averaging, sorting, grouping, cross-referencing, etc. Rather than have each computer

application implement these from scratch, they can rely on the DBMS to supply such calculations.

vi. Change and access logging

Often one wants to know who accessed what attributes, what was changed, and when it was changed. Logging services allow this by keeping a record of access occurrences and changes.

vii. Automated optimization

If there are frequently occurring usage patterns or requests, some DBMS can adjust themselves to improve the speed of those interactions. In some cases the DBMS will merely provide tools to monitor performance, allowing a human expert to make the necessary adjustments after reviewing the statistics collected.

3.5 Meta-data repository

Metadata is data describing data. For example, a listing that describes what attributes are allowed to be in data sets is called "meta-information". The meta-data is also known as data about data.

Activity B

i. **list and explain the components of DBMS**
ii. **mention four types of modelling techniques**

3.6 DBMS Current Trends

As of 1998 database management was in need of new style databases to solve current database management problems. Researchers realized that the old trends of database management were becoming too complex and there was a need for automated configuration and management. Surajit Chaudhuri, Gerhard Weikum and Michael Stonebraker, were the pioneers that dramatically affected the thought of database management systems. They believed that database management needed a more modular approach and that there are so many specifications needs for various users. Since this new development process of database management we currently have endless possibilities. Database management is no longer limited to “monolithic entities”. Many solutions have developed to satisfy individual needs of users. Development of numerous database options has created flexible solutions in database management. Today there are several ways database management has affected the technology world as we know it. Organizations demand for directory services has become an extreme necessity as organizations grow. Businesses are now able to use directory services that provided prompt searches for their company information. Mobile devices are not only able to store contact information of users but have grown to bigger capabilities. Mobile technology is able to cache large information that is used for computers and is able to display it on smaller devices. Web searches have even been affected with database management. Search engine queries are able to locate data within the World Wide Web. Retailers have also benefited from the developments with data warehousing. These companies are able to record customer transactions made within their business. Online transactions have become tremendously popular with the e-business world. Consumers and businesses are able to make payments securely on company websites. None of these current developments would have been possible without the evolution of database management. Even with all the progress and current trends of database management, there will always be a need for new development as specifications and needs grow.

3.7 Examples of Database Management Systems

i. Adabas
ii. Adaptive Server Enterprise
iii. Alpha Five
iv. Computhink's ViewWise
v. CSQL
vi. Daffodil DB
vii. DataEase
viii. FileMaker
ix. Firebird
x. Glom
xi. IBM DB2
xii. IBM UniVerse
xiii. Ingres
xiv. Informix
xv. InterSystems Caché
xvi. Kexi
xvii. WX2
xviii. Linter SQL RDBMS
xix. Mark Logic
xx. Microsoft Access
xxi. Microsoft SQL Server

4.0 Conclusion

For easy access of data and effective information system management dbms is inevitable. This has led to consistent development in the field of database management.

5.0 Summary

i. A DBMS is a set of software programs that controls the organization, storage, management, and retrieval of data in a database.

ii. Databases have been in use since the earliest days of electronic computing.

iii. 1998 marked a new vista in database management, this led to the advent of flexible and easy to use database systems available today.

6.0 Tutor Marked Assignment

i. what is data structure

ii. explain what you understand by the following:

a. Database query language
b. Transaction mechanism

REFERENCES

1. *Free On-line Dictionary of Computing, which is licensed under the GFDL.*
2. Codd, E.F. (1970)."A Relational Model of Data for Large Shared Data Banks". In: *Communications of the ACM* 13 (6): 377–387.
3. itl.nist.gov (1993) *Integration Definition for Information Modeling (IDEFIX)*. 21 December 1993.
4. Seltzer, M. (2008, July). Beyond Relational Databases. Communications of the ACM, 51(7), 52-58.
5. Association for Computing Machinery SIGIR Forum archive Volume 7, Issue 4.
6. Thomas Haigh, "'A Veritable Bucket of Facts:' Origins of the Data Base Management System," ACM SIGMOD Record 35:2 (June 2006).
7. http://en.wikipedia.org/wiki/Database_management_system

1.0

1.0 Introduction

Data-flow diagrams were invented by Larry Constantine, the original developer of structured design, based on Martin and Estrin's "data-flow graph" model of computation. DFDs are used for the visualization of data processing (structured design). It provides no information about timing of processes. Data-flow diagrams (DFDs) are one of the three essential perspectives of the 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 data-flow diagram, users are able to visualize how the system will operate, what the system will accomplish, and how the system will be implemented. DFDs make it possible for the old system's dataflow diagrams to be compared with the new system's data-flow diagrams to draw comparisons to implement a more efficient system.

2.0 Objectives

By the end of this unit the student is expected to be able to:

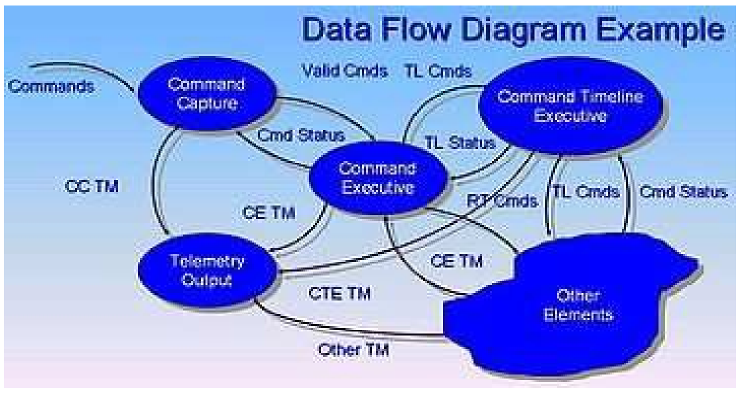
i. Draw a data flow diagram of proposed system
ii. Compare the DFDs of the old and new system to enhance effective performance
iii. Have a good mastery of different DFDs approaches

3.0 DEFINITION

A data-flow diagram (**DFD**) is a graphical representation of the flow of data through an information system. DFDs can also be used for the visualization of data processing (structured design). On a DFD, data items flow from an external data source or an internal data store to an internal data store or an external data sink, *via* an internal process.

A DFD provides no information about the timing or ordering of processes, or about whether processes will operate in sequence or in parallel. It is therefore quite different from a flowchart, which shows the flow of control through an algorithm, allowing a reader to determine what operations will be performed, in what order, and under what circumstances, but not what kinds of data will be input to and output from the system, nor where the data will come from and go to, nor where the data will be stored (all of which are shown on a DFD).

Data Flow Diagram example **From Wikipedia, the free encyclopedia**



It is common practice to draw a context-level data flow diagram first, which shows the interaction between the system and external agents which act as data sources and data sinks. On the context diagram (also known as the *Level 0 DFD*) the system's interactions with the outside world are modelled purely in terms of data flows across the *system boundary*. The context diagram shows the entire system as a single process, and gives no clues as to its internal organization.

This context-level DFD is next "exploded", to produce a Level 1 DFD that shows some of the detail of the system being modeled. The Level 1 DFD shows how the system is divided into sub-systems (processes), each of which deals with one or more of the data flows to or from an external agent, and which together provide all of the functionality of the system as a whole. It also identifies internal data stores that must be present in order for the system to do its job, and shows the flow of data between the various parts of the system.

Data-flow diagrams were invented by Larry Constantine, the original developer of structured design, based on Martin and Estrin's "data-flow graph" model of computation.

Data-flow diagrams (DFDs) are one of the three essential perspectives of the 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 data-flow diagram, users are able to visualize how the system will operate, what the system will accomplish, and how the system will be implemented. The old system's dataflow diagrams can be drawn up and compared with the new system's data-flow diagrams to draw comparisons to implement a more efficient system. Data-flow 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 data-flow diagram.

In the course of developing a set of levelled data-flow diagrams the analyst/designers is forced to address how the system may be decomposed into component sub-systems, and to identify the transaction data in the data model.

There are different notations to draw data-flow diagrams, defining different visual representations for processes, data stores, data flow, and external entities.

3.1The Notation

DFDs show the passage of data through the system by using 5 basic constructs: Data flows, Processes, Data Stores, External Entities, and Physical Resources.

3.1.1 Data Flows

A data flow shows the flow of data from a source to a destination. The flow is shown as an arrowed line with the arrowhead showing the direction of flow. Each data flow should be uniquely identified by a meaningful descriptive name (caption).

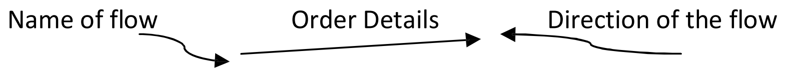


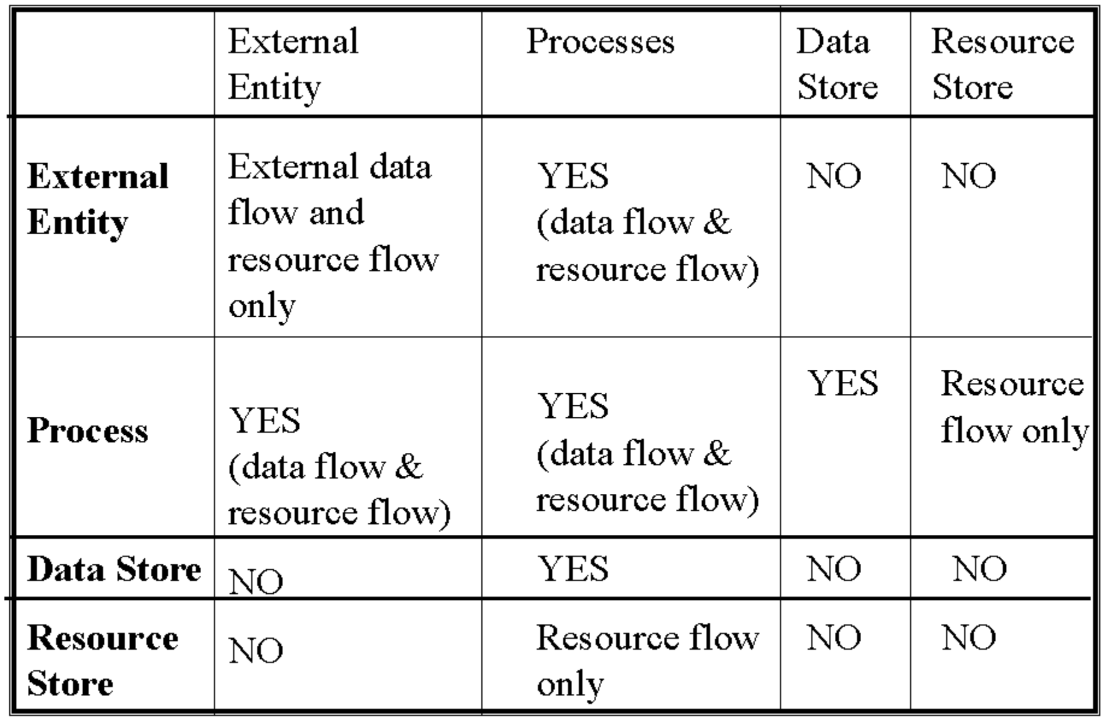
Figure: the notation for a data flow.

Flow may move from an external entity to a process, from a process to another process, into and out of a store from a process, and from a process to an external entity. Flows are **not permitted** to move directly from an external entity to a store or from a store directly to an external entity.

It is generally unacceptable to have a flow moving directly from one external entity to another. However, if it is felt useful to show such a flow, and they do not clutter the diagram, they can be shown as dotted lines.

No two data flows should have the same name. The name of the flows moving in and out of stores may be omitted if the name of the store implies the name of the flow. It is useful to use a name if the flow is especially significant or it is not easy to discern the name of the flow just by examining the diagram. However, omission of names can be justified only in the case of complex diagrams, or when extra long names seem to clutter the diagram. It is good practice to name all notations represented in the diagram. It may be possible to give a combined name for circumstances where many flows move between the same sources and destination.

It is very important that the direction of flow is represented correctly in the diagram. A flow is always from or into a process. The figure below shows the connections, which are allowed and not allowed when constructing a DFD.



	External Entity	Processes	Data Store	Resource Store
External Entity	External data flow and resource flow only	YES (data flow & resource flow)	NO	NO
Process	YES (data flow & resource flow)	YES (data flow & resource flow)	YES	Resource flow only
Data Store	NO	YES	NO	NO
Resource Store	NO	Resource flow only	NO	NO

Permitted Connections of DFD Components

Skidmore et al., p37

3.1.2 Processes

Processes are transformations, changing incoming data flows into outgoing data flows. Processes are drawn as rectangular boxes with a descriptive name occupying the middle of the box. The box has a top stripe that contains an identification number in the left, and the location (or the role carrying out the work) on the right (this is optional and used only in the current physical DFD).

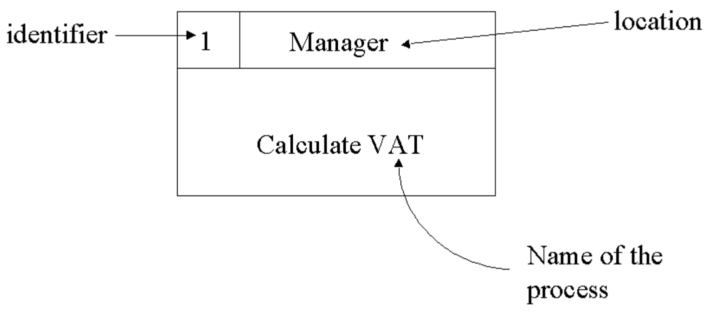


Figure: The notation for Process

The numbering generally follows a left to right convention. This does not indicate priority or sequence. The identification number is purely an identifier. It also helps to associate a high level process with its decomposed subprocesses. This will be made clear to you when we discuss process decomposition. The name of the process should describe what happens to the data as it passes through it. An active verb (verify, compute, extract, create, retrieve, store, determine, etc.) followed by an object or object clause is a suggested notation. In the current physical DFD, the location of the process is placed in the right top box. This might be a physical location or the staff responsible.

3.**1.3 Data Stores**

A store is a repository of data; it may be a card index, a database file, a temporary pile of sales orders awaiting processing, or a folder in a filing cabinet. The store may contain permanent data or temporary accumulations (pending documents, daily movements). A store is represented by an open-ended box and is given a meaningful descriptive name. Each store is also given a reference number prefixed by a letter. In current physical DFDs manual data stores are shown using the letter 'M', and a 'D' used to represent a computer data In contrast to these permanent data stores, data can also be held for a short time in temporary or transient data stores. These are identified by a 'T'. If they are also manual then a 'T(M)' is used.

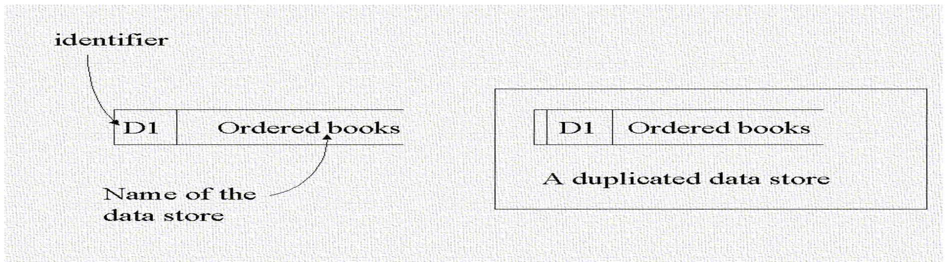


Figure: the notation for 'data store'.

In logical and required system DFD, data stores are regarded as computerised and hence only a 'D' will be used. Some transient stores may remain and retain the 'T'. To prevent a DFD becoming 'spider's web' of crossing lines, the same data store may be included more than once on a DFD. Such duplication is shown by an additional vertical line within the store symbol.

3.1.4 Direction of Flow

If the arrow from the store is single headed and points towards the process, this signifies a 'read' action. In other words, the process does not alter the contents of the store, it only access the data available. For example, the flow from the data store 'Customers' in the figure below.

If the single arrow head points towards the data store then, this indicates a 'write' action, e.g. creating a record. The flow to the data store, "'Hold' forms" is an example of a write action.

An 'update' will consist of both a read and a write. This could be shown either by a double-headed arrow or 2 single arrows on either direction.

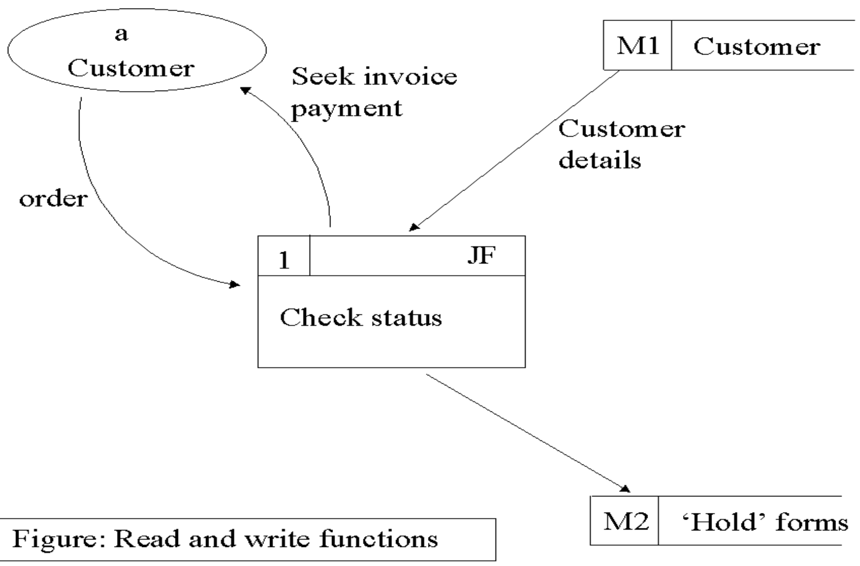
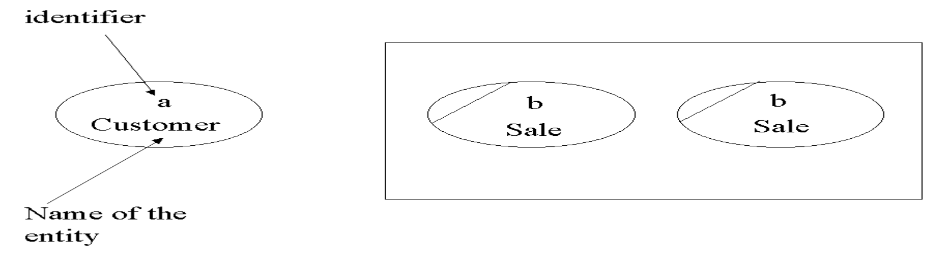


Figure: Read and write functions

3.1.5 External Entities (Source or Sink)

The external entity represents a person or a part of an organisation which sends or receives data from the system but considered to be outside the system boundary (scope of the project). As with the data stores these may be duplicated on a DFD to simplify presentation. External entities may be further referenced by the use of an alpha character, and this is particularly recommended if at a lower level the entity is being decomposed.



An external entity Duplicated external entities

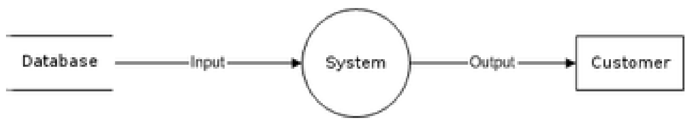
Sometimes external entities are referred to as *sources* and *sinks*. An External entity either supplies data to the system, which makes it a source and /or receives data from the system, which makes it a sink.

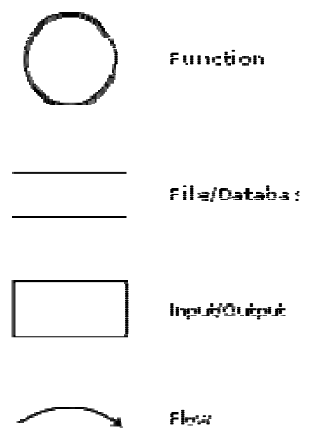
3.1.6 Physical Resources

A physical flow represents the flow of material (as opposed to data flows representing the flow of information), the movement of some resources or goods which are relevant to the information system, from source to destination. They are included to aid communication. A physical flow is represented by a broad arrow. The resource store is represented by a closed rectangle. Physical flows add clutter to the DFD by their physical size. However they can be useful for:

i. Showing significant resource flows and states. This representation is often more meaningful to users than logical data flows which may appear a little abstract.
ii. Getting started in a project. Users may describe the system in terms of physical flows and stores.
iii. Finally and importantly, the physical resources may actually be what the system is all about. It is certainly equally important to send both the goods and the invoice. The practical aspects of the system may be lost to an analyst who concentrates too much on the neatness of the data flow.

3.2 Developing A Data-Flow Diagram

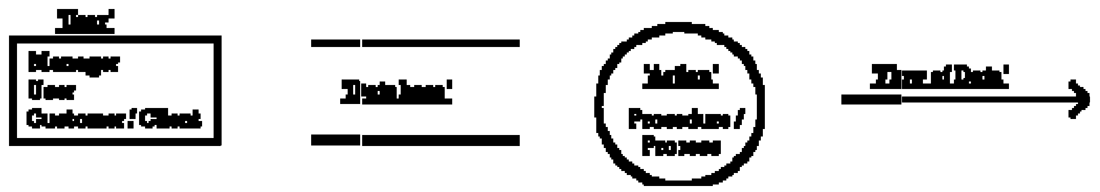




Data-flow diagram - Yourdon/DeMarco notation

When examining an existing Information System or analysing the Information that is going to be designed, it is important to recognise *what* the data is, *where* the data comes from, *how* it passes from one point to another within the Information System and finally how it will be *used* by the intended audience or user. A Data Flow Diagram is one of the best ways to illustrate this. Certain conventions are used that we will follow.

Symbols used:



To create a data flow diagram, it is best to collect together the required information before you start.

Examples

(A) Finding / Using a book for research in a Resource Centre. Let us produce a data flow diagram to illustrate the procedure described above

i. Decide on the probable users that may be involved in the procedure; in this case it is very easy as it is just the **student** borrower.

ii. Determine which Data bases the user(s) may need to interact with; again fairly simple in this case, let us assume the **Oracle system** and the **Books in the shelves**.

iii. Decide on the probable 'Actions' that will need to be carried out to complete the procedure. It is not necessary to be over enthusiastic but do allow for possible additions as your diagram progresses.

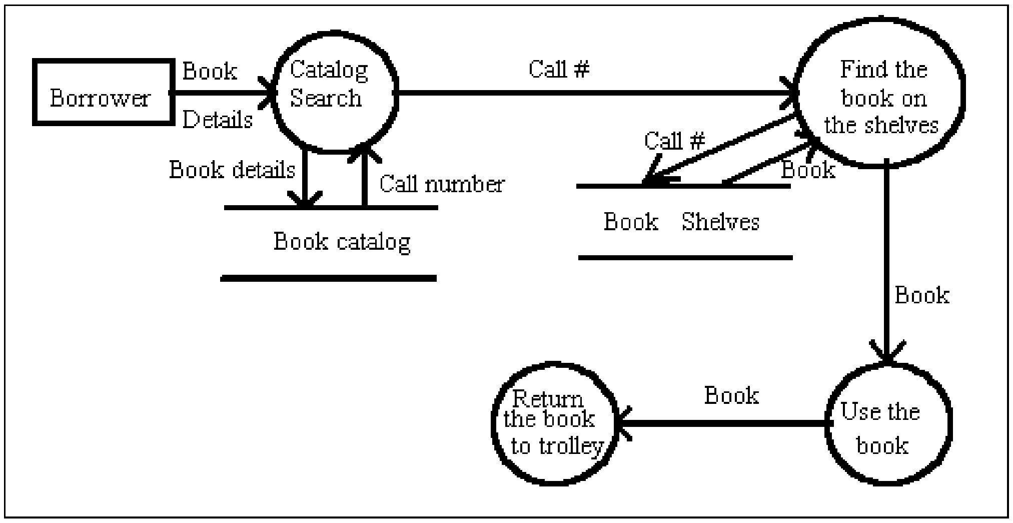
a. *Searching the catalog* for the required book,
b. *Retrieving the book* from the shelves,

c. *Using the book* to conduct the required research
d. *Returning the book* to the shelves or trolley

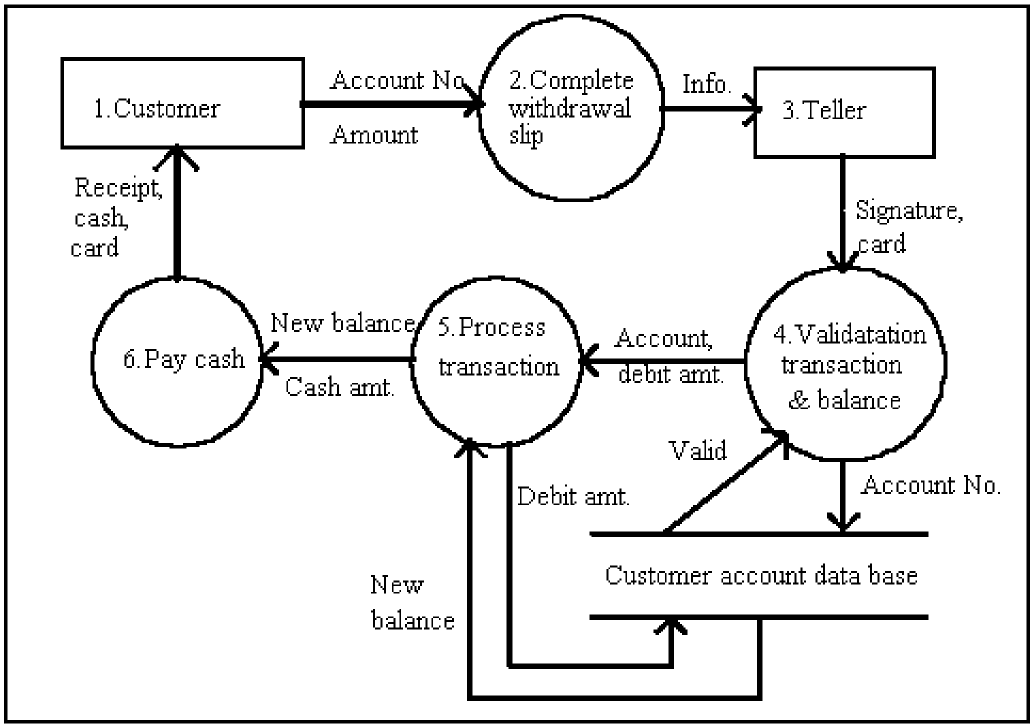
iv. work out the data that is going to be needed to carry out each of these actions and then just put them together like a jigsaw puzzle:

a. Book title / Author / Subject matter to be able to search the catalog;
b. Retrieved Call # to be able to find the book in the shelves;
c. The book itself to be able to conduct the research and finally put it back.

Diagrammatically this is what you have:



(B) following the above this is what you have for Cash withdrawal from a bank.



A Data Flow Diagram (DFD) is a diagrammatic representation of the information flows within a system, showing:

i. how information enters and leaves the system,
ii. what changes the information,
iii. where information is stored.

In SSADM a DFD model includes supporting documentation describing the information shown in the diagram. DFDs are used not only in structured system analysis and design, but also as a general process modelling tool. There are a number of commercial tools in the market today which are based on DFD modelling. SSADM uses DFDs in three stages of the development process:

i. Current Physical DFDs. These record the results of conventional fact finding.
ii. Current Logical DFDs. The logical information processing of the current system
iii. Required Logical DFDs. The logical information processing requirements of the proposed system.

Activity A

i. What do understand by DFDs
ii. Draw a typical DFDs
iii. With the aid of diagram describe each of the Yourdon DFD notations

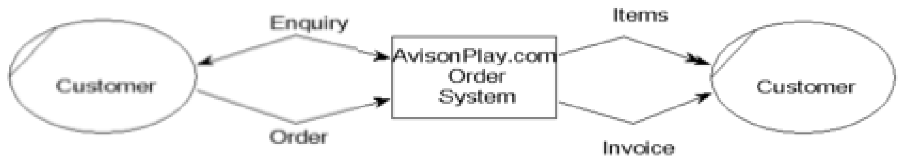
3.2.1 Top-Down Approach

i. The system designer makes "a context level DFD" or Level 0, which shows the "interaction" (data flows) between "the system" (represented by one process) and "the system environment" (represented by terminators).
ii. The system is "decomposed in lower-level DFD (Level 1)" into a set of "processes, data stores, and the data flows between these processes and data stores".
iii. Each process is then decomposed into an "even-lower-level diagram containing its subprocesses".
iv. This approach "then continues on the subsequent subprocesses", until a necessary and sufficient level of detail is reached which is called the primitive process (aka chewable in one bite).

DFD is also a virtually designable diagram that technically or diagrammatically describes the inflow and outflow of data or information that is provided by the external entity.

3.2.2 Event Partitioning Approach

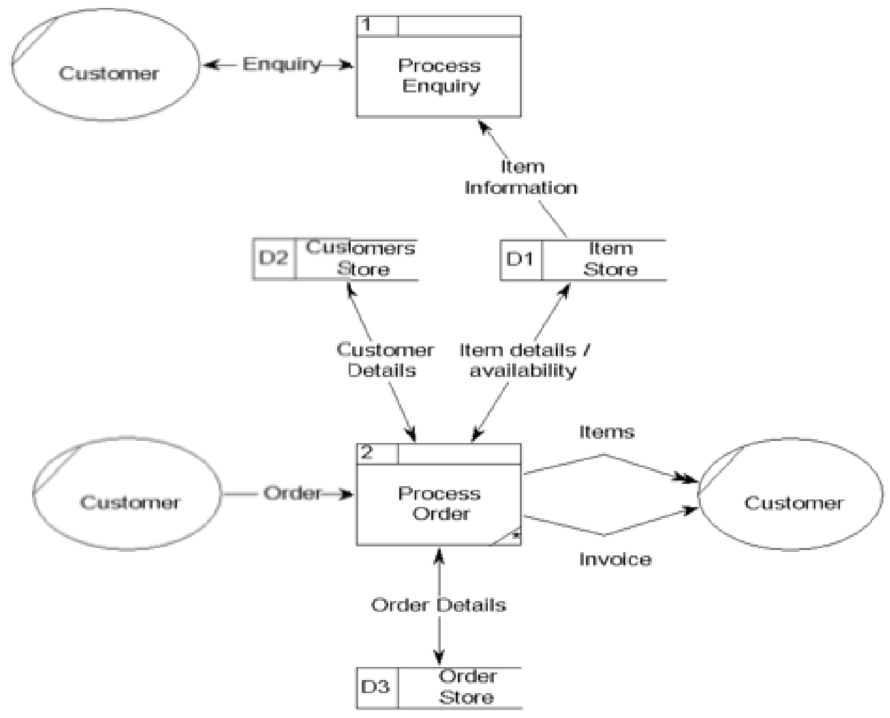
Event partitioning was described by Edward Yourdon in *Just Enough Structured Analysis*.



A context level Data flow diagram created using Select SSADM.

This level shows the overall context of the system and its operating environment and shows the whole system as just one process. It does not usually show data stores, unless they are "owned" by external systems, e.g. are accessed by but not maintained by this system, however, these are often shown as external entities.

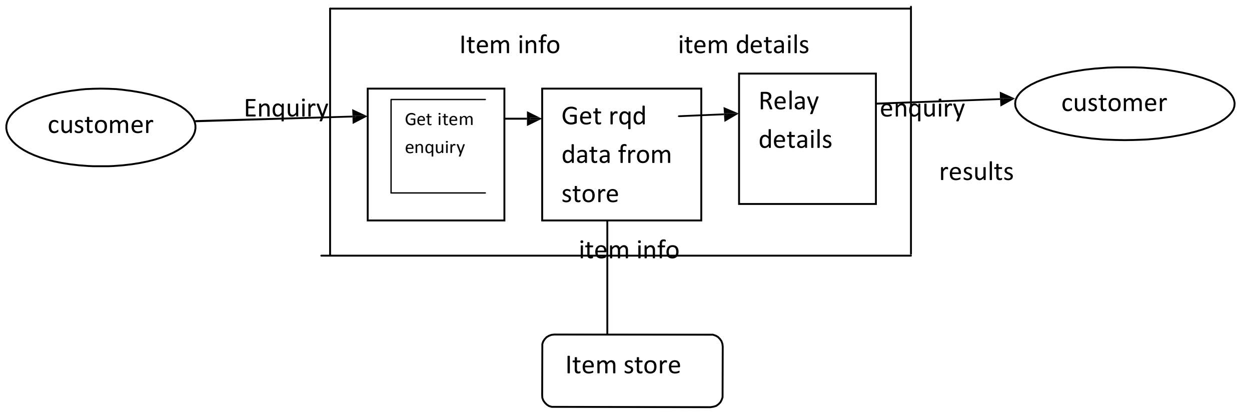
Level 1 (High Level Diagram)



A Level 1 Data flow diagram for the same system.

This level (level 1) shows all processes at the first level of numbering, data stores, external entities and the data flows between them. The purpose of this level is to show the major high-level processes of the system and their interrelation. A process model will have one, and only one, level-1 diagram. A level-1 diagram must be balanced with its parent context level diagram, i.e. there must be the same external entities and the same data flows, these can be broken down to more detail in the level 1, e.g. the "inquiry" data flow could be split into "inquiry request" and "inquiry results" and still be valid.

Level 2 (Low Level Diagram)



A Level 2 Data flow diagram showing the "Process Enquiry" process for the same system.

This level is a decomposition of a process shown in a level-1 diagram, as such there should be a level-2 diagram for each and every process shown in a level-1 diagram. In this example processes 1.1, 1.2 & 1.3 are all children of process 1, together they wholly and completely describe process 1, and combined must perform the full capacity of this parent process. As before, a level-2 diagram must be balanced with its parent level-1 diagram.

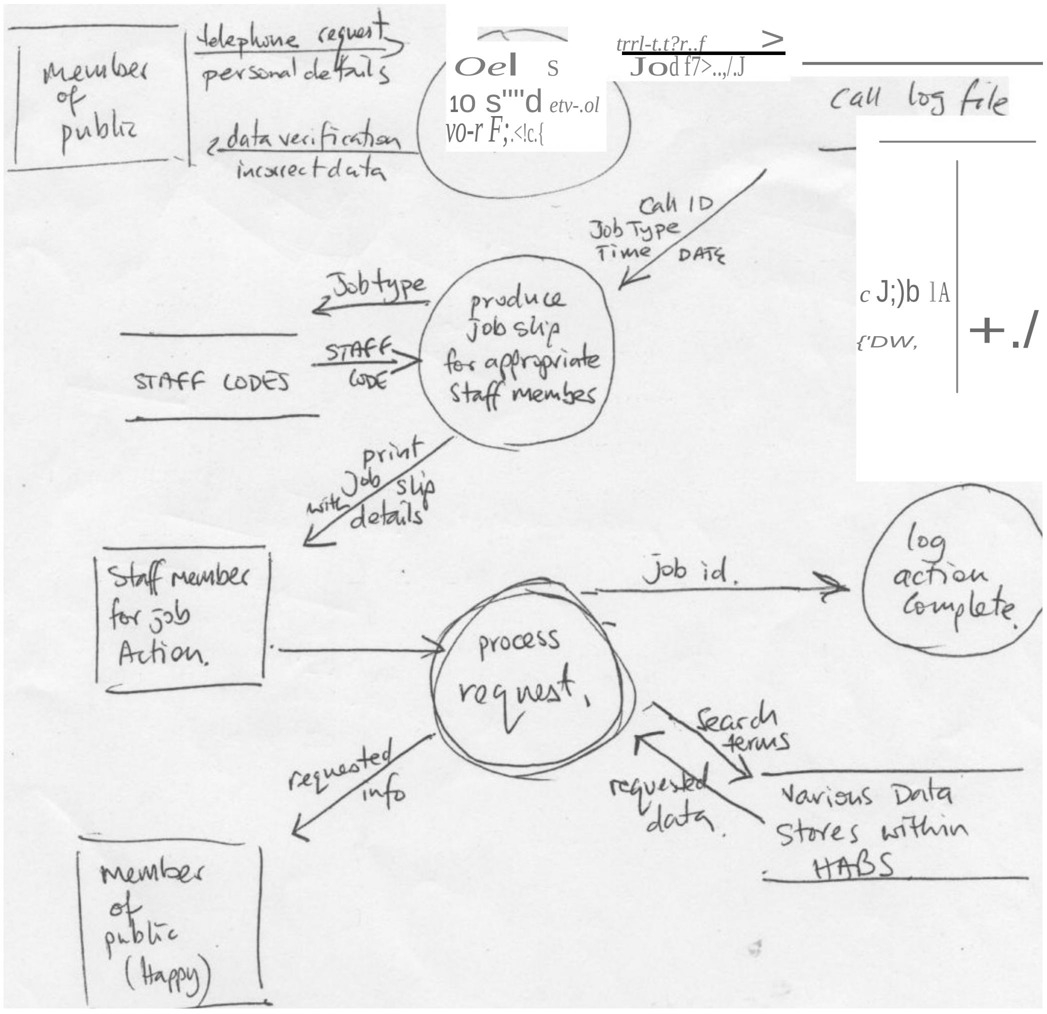
The examples given in this unit are somewhat simplistic. Take a good consideration about a call logging system for a school telephone switchboard - In this system, the problem is taking all calls and, when they can't be patched through to the right person, logging the time of call, caller, etc, who the call was for and what it was about. This would allow the operator to generate a notice for the intended, albeit absent, recipient of the call.

Below are two DFDs for this system - notice how we don't have to have an 'all in one' DFD... you can break your DFDs into parts if appropriate - sometimes trying for a 'master DFD' just doesn't work and only complicates matters!

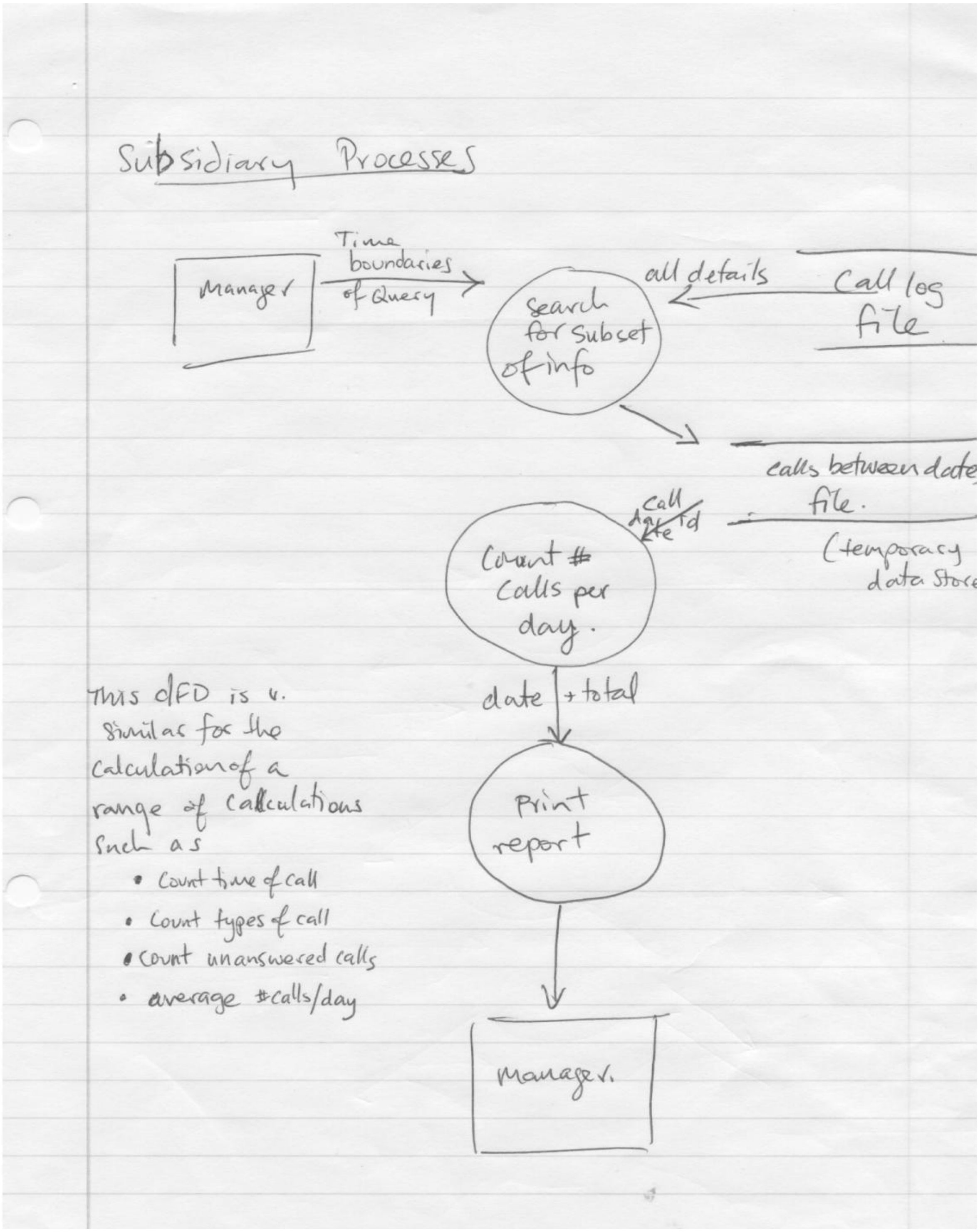
Core Process:

C U to 1 @ U*{)

CeJ-(t '?ioCJLS



Subsidiary Process



Subsidiary Processes

This DFD is v. similar for the calculation of a range of calculations such as

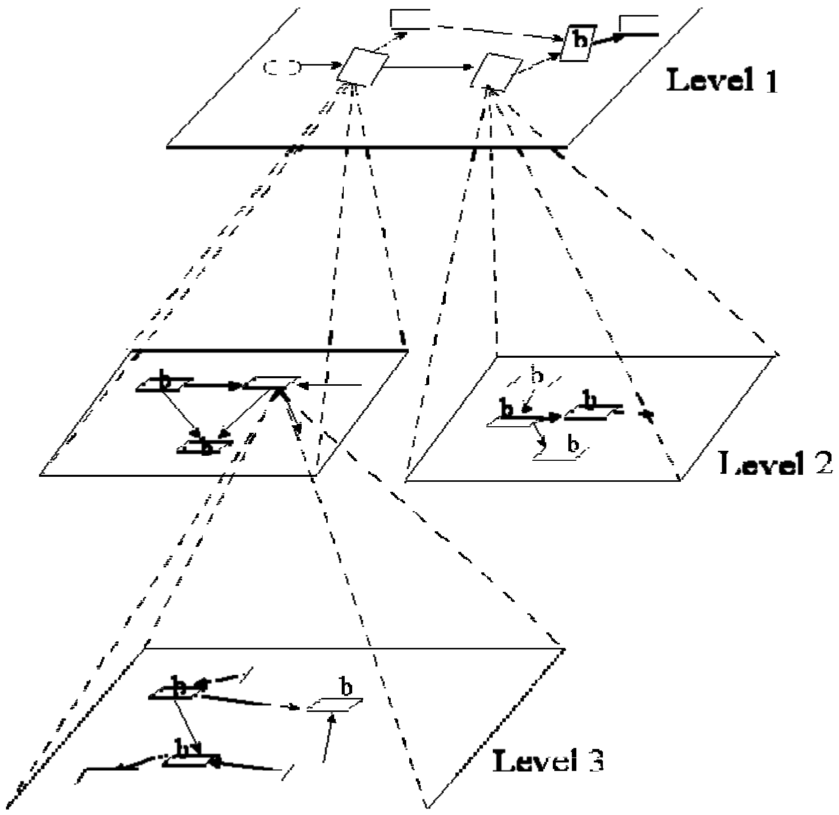
- Count time of call
- Count types of call
- Count unanswered calls
- average #calls/day

3.3 Modelling Hierarchy

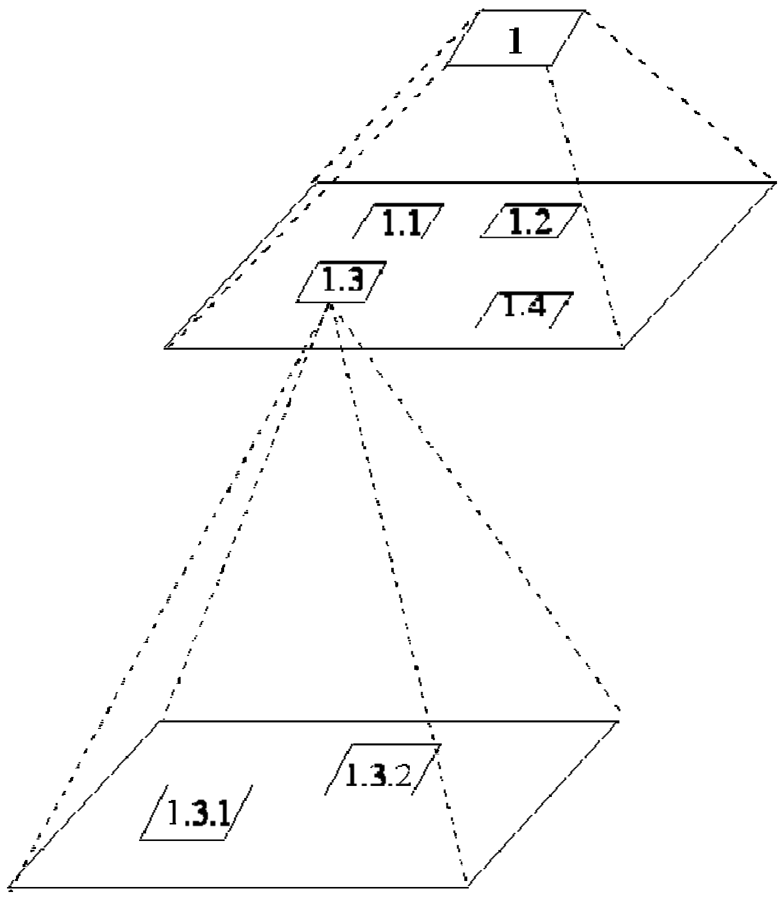
A major advantage of a DFD is its use in communication between user and analyst, or even between 2 analysts. A DFD becomes difficult to understand when it has more than 7-9 processes. If there is a tendency to overstep this (in other words, if the modeller feels the figure is too complex for easy understanding) then the DFD should be redrawn with processes that are logically grouped together being replaced by a single process to encompass them all. The processes which were replaced should appear on another DFD (which is considered to be at a lower level) that shows how this combined process can be exploded into its constituents. These constituents themselves may be complex and can be broken down into sub processes shown on a DFD at a lower level. This is known as decomposing the DFD.

The DFD that shows the entire system within a single diagram is the top-level or ‘level 1’ DFD. The DFD that are expansions of processes at the top-level are ‘level 2’ DFDs. Levels below this are called ‘level 3’, level 4’, etc. Processes that are not further decomposed are bottom-level processes. Processes from the top-level DFD may be broken down (decomposed) into a number of levels if they are complex, or may be not broken down at all if they are simple. Thus, it is possible to have bottom-level processes appearing at all levels of the DFD.

In the figure below, the bottom-level processes are denoted by the letter ‘b’.



If a process is decomposed, the identifiers of the lower-level processes are prefixed by the identifier of the higher-level process. For example, if process 1 is decomposed, then the lower-level processes will be identified as 1.1, 1.2, etc. Similarly, if process 1.3 is subsequently decomposed, the lower-level processes will be 1.3.1, 1.3.2, and so on. This is shown in the figure below.



Note that all of the data flows to and from the high-level process have to be represented at the lower level. They can be either duplicated or broken down to several flows. If new data flows are identified at the lower level which cross the frame (indicating they are not internal to the process), these should be reflected at the higher level so that consistency is maintained between the levels.

This concept can also be extended backwards where the complete level 0 DFD is a one process diagram which summarises the inputs and the outputs of the system under consideration. This is called the *context diagram*.

3.3.1 **Advantages of decomposition**

i. Provide ease of understanding.
ii. It falls naturally into line with analyst's top down approach to decomposition.
iii. The various levels represent the various degrees of detail by which the system is represented. This is very useful during discussions with users, either in factfinding or getting agreement about system specification. Different users may want to view the system at different levels of detail.

iv. By the incorporation of different levels, the DFD can provide the view of the whole system or of an area of interest.

3.4 Constructing the Logical Data Flow Diagram

The first step is to read carefully the specification looking for and listing all mentions of data the system is to handle. Some data originates in the environment and is supplied as input documents to the system. Some data is generated by the program and delivered as output documents. Some data is retrieved from or saved in data stores.

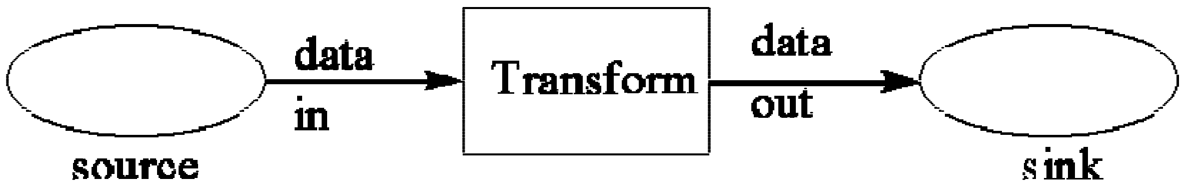
Hint: When identifying data implied by a specification look out for nouns.

The next step is to list all mention of processing that the data undergo.

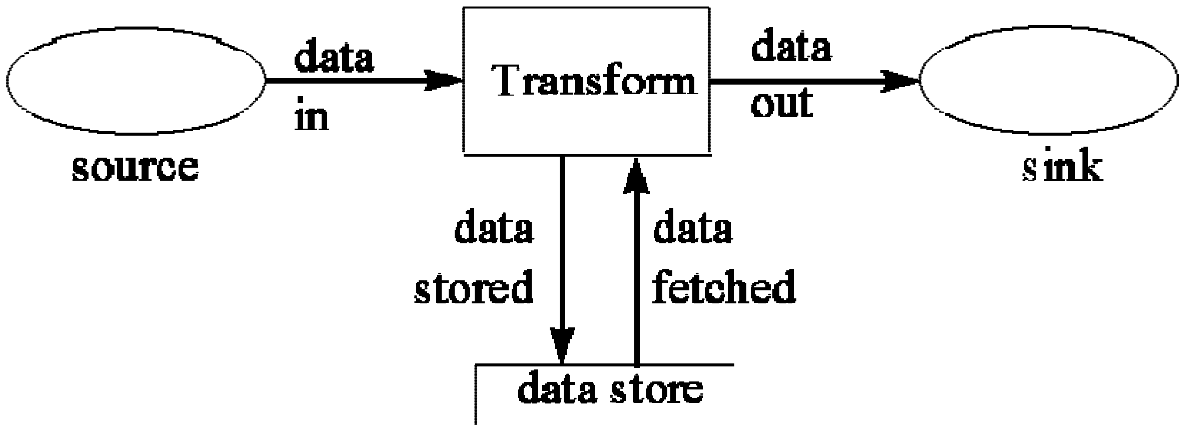
Hint: When doing this look out for verbs.

Now you can begin to develop a data flow diagram.

The figure below shows the simplest data flow diagram.



The figure above shows a top-level description of a system specification. The system as a whole is viewed as one process. The input and output to the system at this level of abstraction is from the environment. In the above figure there is a single external entity (source) which sends data (input) to the system, and a single external entity (sink) which receives data from the system (output). This is commonly known as a ‘context diagram’, or a 0-level DFD. If the system also updates an external data store (e.g. a database, a file, a record) then the context diagram will look like:



An internal data store would not be shown at this level of abstraction and would appear only in the subsequent refinements of the transform.

3.5 Naming Convention

The name of a notation is usually written within the symbol. Choose brief verb phrases for processes and noun phrases for data flows. It is important that the name should say only what is necessary. Do not describe the representation of the data, its recording medium, or its type; or how the transforms are implemented - say only what processing is to be done.

3.5.1 Hints on names on DFDs

Data Flows

i. Name the data + adjective(s)
ii. Say what is known about it (e.g. valid account number)

Processes

i. The name is a command: verb + nouns (e.g. validate account number) or

verb + object/object phrase

ii. **Do not use** 'and', 'or', 'then', 'if', 'repeat', or any other words that specify control flow.

Data Stores: Show only net flow in/out/both (i.e. indicate whether it is read-only, write-only, or updated).

3.6. Advantages of DFDs

i. A simple but powerful graphic technique which is easily understood.
ii. Represents an information system from the viewpoint of data movements, which includes the inputs and outputs to which people can readily relate.
iii. The ability to represent the system at different levels of details gives added advantage (you can include the advantages of decomposition listed earlier)
iv. Helps to define the boundaries of the system.
v. A useful tool to use during interviews.
vi. Serve to identify the information services the users require, on the basis of which the future information system will be constructed.

Activity B

i. What are the approaches applied in DFDs
ii. What do you understand by event partitioning approach
iii. Explain the steps involved in top-down approach

3.0 Conclusion

DFD is a graphical representation of the flow of data through an information system.

4.0 Summary

i. DFDs can be broken into component parts since trying to create a master DFD can complicate issues
ii. DFD is also a virtually designable diagram that technically or diagrammatically describes the inflow and outflow of data or information that is provided by the external entity.

5.0 Tutor Marked Assignment

i. Make a large DFD for office attendance management system
ii. Give a detailed explanation on how it will work

7.0 ***Reference***

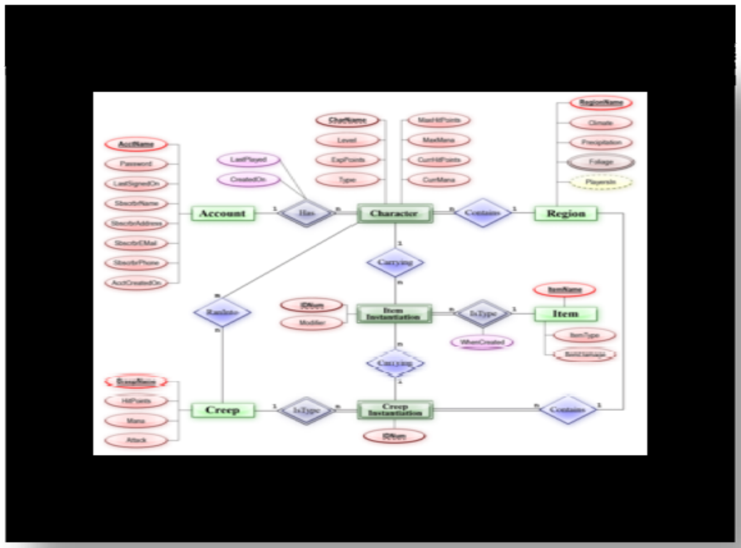
1. S Skidmore, R Framer and G Mills, *SSADM Models and Methods Version 4*.

2. M Goodland and C Slater, SSADM VERSION 4:A Practical Approach, McGraw-Hill, 1995.
3. http://www.thinking-forth.sourceforge.net/
4. "http://en.wikipedia.org/wiki/model"

Contents

1.0 INTRODUCTION

In software engineering, an Entity-Relationship Model (ERM) is an abstract and conceptual representation of data. Entity-relationship modeling is a database modeling method, used to produce a type of conceptual schema or semantic data model of a system, often a relational database, and its requirements in a top-down fashion. Diagrams created using this process are called *entity-relationship diagrams*, or *ER diagrams* or *ERDs* for short. The first stage of information system design uses these models during the requirements analysis to describe information needs or the type of information that is to be stored in a database. The data modeling technique can be used to describe any ontology (i.e. an overview and classifications of used terms and their relationships) for a certain area of interest. In the case of the design of an information system that is based on a database, the conceptual data model is, at a later stage (usually called logical design), mapped to a logical data model, such as the relational model; this in turn is mapped to a physical model during physical design. Note that sometimes, both of these phases are referred to as "physical design". There are a number of conventions for entity-relationship diagrams (ERDs). The classical notation mainly relates to conceptual modeling. There are a range of notations employed in logical and physical database design, such as IDEF1X.



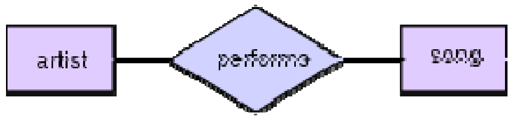
A sample ER diagram From Wikipedia, the free encyclopedia

2.0 OBJECTIVE

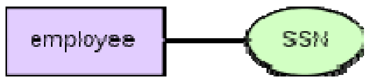
It is expected that at the end of this unit, you should:

i. Understand ER Diagrams
ii. be able explain and apply ER models
iii. be conversant with ER connections
iv. understand different types of relationship
v. understand the applications of different layers
vi. Learn the 2-layer style
vii. Understand what a domain model is, and what UML relationships are
viii. Learn the 3-layer style
ix. Learn about object data source controls

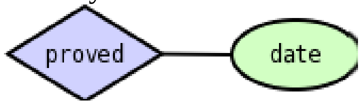
3.0 THE BUILDING BLOCKS: ENTITIES, RELATIONSHIPS, AND ATTRIBUTES



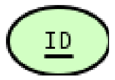
Two related entities



An entity with an attribute



A relationship with an attribute



Primary key

An entity may be defined as a thing which is recognized as being capable of an independent existence and which can be uniquely identified. An entity is an abstraction from the complexities of some domain. When we speak of an entity we normally speak of some aspect of the real world which can be distinguished from other aspects of the real world.

An entity may be a physical object such as a house or a car, an event such as a house sale or a car service, or a concept such as a customer transaction or order. Although the term entity is the one most commonly used, following Chen we should really distinguish between an entity and an entity-type. An entity-type is a category. An entity, strictly speaking, is an instance of a given entity-type. There are usually many instances of an entity-type. Because the term entity-type is somewhat cumbersome, most people tend to use the term entity as a synonym for this term. Entities can be thought of as nouns. Examples: a computer, an employee, a song, a mathematical theorem. Entities are represented as rectangles.

A relationship captures how two or more entities are related to one another. Relationships can be thought of as verbs, linking two or more nouns. Examples: an *owns* relationship between a company and a computer, a *supervises* relationship between an employee and a department, a *performs* relationship between an artist and a song, a *proved* relationship between a mathematician and a theorem. Relationships are represented as diamonds, connected by lines to each of the entities in the relationship. The model's linguistic aspect described above is utilized in the database query language ERROL.

Entities and relationships can both have attributes. Examples: an *employee* entity might have a *Social Security Number* (SSN) attribute; the *proved* relationship may have a *date* attribute. Attributes are represented as ellipses connected to their owning entity sets by a line. Every entity (unless it is a weak entity) must have a minimal set of uniquely identifying attributes, which is called the entity's primary key. Entity-relationship diagrams don't show single entities or single instances of relations. Rather, they show entity sets and relationship sets. Example: a particular *song* is an entity. The collection of all songs in a database is an entity set. The *eaten* relationship between a child and her lunch is a single relationship. The set of all such child-lunch relationships in a database is a relationship set. In other words, a relationship set corresponds to a relation in mathematics, while a relationship corresponds to a member of the relation. Certain cardinality constraints on relationship sets may be indicated as well.

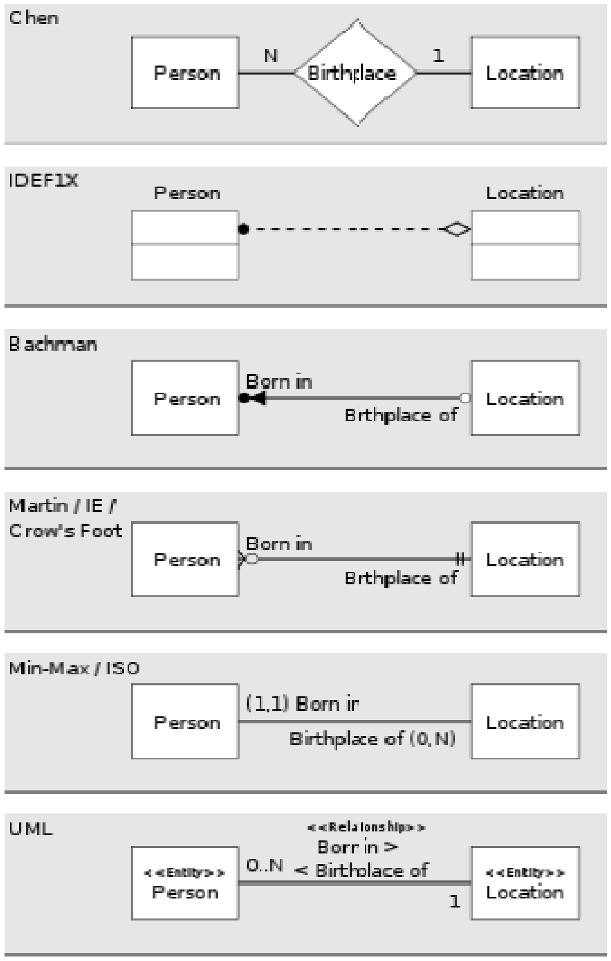
3.1 DIAGRAMMING CONVENTIONS

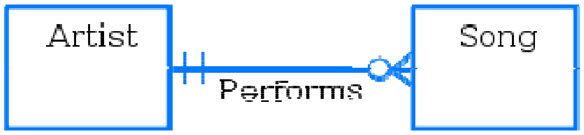
Entity sets are drawn as rectangles, relationship sets as diamonds. If an entity set participates in a relationship set, they are connected with a line. Attributes are drawn as ovals and are connected with a line to exactly one entity or relationship set.

3.1.1 Cardinality constraints are expressed as follows:

i. a double line indicates a *participation constraint*, i.e. totality or surjectivity: all entities in the entity set must participate in *at least one* relationship in the relationship set;
ii. an arrow from entity set to relationship set indicates a key constraint, i.e. injectivity: each entity of the entity set can participate in *at most one* relationship in the relationship set;
iii. a thick line indicates both, i.e. bijectivity: each entity in the entity set is involved in *exactly one* relationship.
iv. an underlined name of an attribute indicates that it is a key: two different entities or relationships with this attribute always have different values for this attribute.

Attributes can really clutter up the diagram, so they are often omitted; related diagram techniques often list entity attributes within the rectangles drawn for entity sets.





Two related entities shown using Crow's Foot notation

Chen's notation for entity-relationship modelling uses rectangles to represent entities, and diamonds to represent relationships. This notation is appropriate because Chen's relationships are first-class objects: they can have attributes and relationships of their own.

3.2 THE ENTITY-RELATIONSHIP MODEL

The **E-R** (entity-relationship) data model views the real world as a set of basic **objects** (entities) and **relationships** among these objects. It is intended primarily for the DB design process by allowing the specification of an **enterprise scheme**. This represents the overall logical structure of the DB.

3.2.1 ENTITIES AND ENTITY SETS

i. An **entity** is an object that exists and is distinguishable from other objects. For instance, Okoro Okonkwo with S.I.N. 890-12-3456 is an entity, as he can be uniquely identified as one particular person in the universe.
ii. An entity may be **concrete** (a person or a book, for example) or **abstract** (like a holiday or a concept).
iii. An **entity set** is a set of entities of the same type (e.g., all persons having an account at a bank).
iv. Entity sets **need not be disjoint**. For example, the entity set *employee* (all employees of a bank) and the entity set *customer* (all customers of the bank) may have members in common.
v. An entity is represented by a set of **attributes**.
   a. E.g. name, S.I.N., street, city for ``customer'' entity.
   b. The **domain** of the attribute is the set of permitted values (e.g. the telephone number must be seven positive integers).
vi. Formally, an attribute is a **function** which maps an entity set into a domain.
   a. Every entity is described by a set of (attribute, data value) pairs.
   b. There is one pair for each attribute of the entity set.
   c. E.g. a particular *customer* entity is described by the set {(name, Okoro), (S.I.N., 890-123-456), (street, Iweka), (city, Owerri)}.

An analogy can be made with the programming language notion of Type definition.

i. The concept of an **entity set** corresponds to the programming language **type definition**.
ii. A variable of a given type has a particular value at a point in time.
iii. Thus, a programming language variable corresponds to an **entity** in the E-R model.

The figure above shows two entity sets. We will be dealing with five entity sets in this section:

i. *branch*, the set of all branches of a particular bank. Each branch is described by the attributes *branch-name*, *branch-city* and *assets*.
ii. *customer*, the set of all people having an account at the bank. Attributes are *customer-name*, *S.I.N.*, *street* and *customer-city*.
iii. *employee*, with attributes *employee-name* and *phone-number*.
iv. *account*, the set of all accounts created and maintained in the bank. Attributes are *account-number* and *balance*.
v. *transaction*, the set of all account transactions executed in the bank. Attributes are *transaction-number*, *date* and *amount*.

3.2.2 RELATIONSHIPS AND RELATIONSHIP SETS

A **relationship** is an association between several entities.

A **relationship set** is a set of relationships of the same type.

Formally it is a mathematical relation on $n \geq 2$(possibly non-distinct) sets.

If $E_1, E_2, \ldots, E_n$are entity sets, then a relationship set R is a **subset** of

$$\{(e_1, e_2, \ldots, e_n) \mid e_1 \in E_1, e_2 \in E_2, \ldots, e_n \in E_n\}$$

where $(e_1, e_2, \ldots, e_n)$is a relationship.

For example, consider the two entity sets *customer* and *account*. We define the relationship *CustAcct* to denote the association between customers and their accounts. This is a **binary** relationship set. Going back to our formal definition, the relationship set *CustAcct* is a subset of all the possible customer and account pairings. This is a binary relationship. Occasionally there are relationships involving more than two entity sets.

The **role** of an entity is the function it plays in a relationship. For example, the relationship *works-for* could be ordered pairs of *employee* entities. The first employee takes the role of manager, and the second one will take the role of worker.

A relationship may also have **descriptive** attributes. For example, *date* (last date of account access) could be an attribute of the *CustAcct* relationship set.

3.2.3 ATTRIBUTES

It is possible to define a set of entities and the relationships among them in a number of different ways. The main difference is in how we deal with attributes.

i. Consider the entity set *employee* with attributes *employee-name* and *phone-number*.
ii. We could argue that the phone be treated as an entity itself, with attributes *phone-number* and *location*.
iii. Then we have two entity sets, and the relationship set *EmpPhn* defining the association between employees and their phones.
iv. This new definition allows employees to have several (or zero) phones.
v. New definition may more accurately reflect the real world.
vi. We cannot extend this argument easily to making *employee-name* an entity.

The question of what constitutes an entity and what constitutes an attribute depends mainly on the structure of the real world situation being modeled, and the semantics associated with the attribute in question.

3.2.4 MAPPING CONSTRAINTS

An E-R scheme may define certain constraints to which the contents of a database must conform.

i. **Mapping Cardinalities:** express the number of entities to which another entity can be associated via a relationship. For binary relationship sets between entity sets A and B, the mapping cardinality must be one of:
   1. **One-to-one**: An entity in A is associated with at most one entity in B, and an entity in B is associated with at most one entity in A.
   2. **One-to-many**: An entity in A is associated with any number in B. An entity in B is associated with at most one entity in A.
   3. **Many-to-one**: An entity in A is associated with at most one entity in B. An entity in B is associated with any number in A.
   4. **Many-to-many**: Entities in A and B are associated with any number from each other.

   The appropriate mapping cardinality for a particular relationship set depends on the real world being modeled. (Think about the *CustAcct* relationship...)

ii. **Existence Dependencies:** if the existence of entity X depends on the existence of entity Y, then X is said to be **existence dependent** on Y. (Or we say that Y is the **dominant** entity and X is the **subordinate** entity.) For example,
   a. Consider *account* and *transaction* entity sets, and a relationship *log* between them.
   b. This is one-to-many from account to transaction.
   c. If an *account* entity is deleted, its associated *transaction* entities must also be deleted.
   d. Thus *account* is dominant and *transaction* is subordinate.

3.2.5 KEYS

Differences between entities must be expressed in terms of attributes.

i. A **superkey** is a set of one or more attributes which, taken collectively, allow us to identify uniquely an entity in the entity set. For example, in the entity set *customer*, *customer-name* and *S.I.N.* is a superkey. Note that *customer-name* alone is not, as two customers could have the same name. A superkey may contain extraneous attributes, and we are often interested in the smallest superkey. A superkey for which no subset is a superkey is called a **candidate key**. In the example above, *S.I.N.* is a candidate key, as it is minimal, and uniquely identifies a customer entity.
ii. A **primary key** is a candidate key (there may be more than one) chosen by the DB designer to identify entities in an entity set.

An entity set that does not possess sufficient attributes to form a primary key is called a **weak entity set.** One that does have a primary key is called a **strong entity set**.

For example,

i. The entity set *transaction* has attributes *transaction-number*, *date* and *amount*.
ii. Different transactions on different accounts could share the same number.
iii. These are not sufficient to form a primary key (uniquely identify a transaction).
iv. Thus *transaction* is a weak entity set.

For a weak entity set to be meaningful, it must be part of a one-to-many relationship set. This relationship set should have no descriptive attributes. (Why?) The idea of strong and weak entity sets is related to the existence dependencies seen earlier.

i. Member of a strong entity set is a dominant entity.
ii. Member of a weak entity set is a subordinate entity.

A weak entity set does not have a primary key, but we need a means of distinguishing among the entities.

The **discriminator** of a weak entity set is a set of attributes that allows this distinction to be made. The **primary key of a weak entity set** is formed by taking the primary key of the strong entity set on which its existence depends plus its **discriminator**.

To illustrate:

i. *transaction* is a weak entity. It is existence-dependent on *account*.
ii. The primary key of *account* is *account-number*.
iii. *transaction-number* distinguishes transaction entities within the same account (and is thus the discriminator).
iv. So the primary key for *transaction* would be *(account-number*, *transaction-number)*.

Just Remember: The primary key of a weak entity is found by taking the primary key of the strong entity on which it is existence-dependent, plus the discriminator of the weak entity set.

Activity A/ self assessment exercise

i. Describe the following terms
  a. Entity and entity set
  b. Relationship and relationship set
  c. Attribute and attribute set
ii. Explain fully the constraints to which the contents of a database must
 conform.
iii. Differentiate between weak and strong entity sets

3.3 ENTITY-RELATIONSHIP MODEL

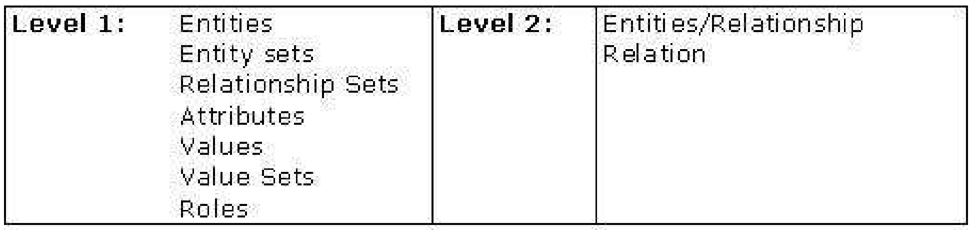
Entity–Relationship (ER) modeling is an important step in information system design and software engineering. Entities and relationships are a natural way to organize physical things as well as information. The ER concept is the basic fundamental principle for conceptual modeling.

The entity-relationship model can be used as a basis for **unification of different views of data**: the **network model**, the **relational model** and the **entity set model**. In the entity-relationship model the semantic of data is much more apparent than in the other models. Also the handling of data ambiguity is much better than for example in the network model with its ambiguous arrows. In the entity-relationship model all kinds of mappings are handled uniformly. Not everything is treated as an entity such as in the entity set model and the entity-relationship model also uses n-ary relationships and not only binary relationships like the entity set model does.

Also a special diagrammatic technique is introduced as a tool for database design – the **entity relationship diagram**. The entitiy-relationship model adopts the more natural view that the real world consists of **entities** and **relationships**.

In this section we introduce the entity-relationship model using a framework of multilevel views of data. There can be identified four levels of views of data. The entity relationship model is mainly concerned with the first two levels. The first level deals with information concerning entities and relationships that exists in our minds. The second level deals with information structure concerning the organization of information in which entities and relationships are represented by data.

3.3.1 INFORMATION CONCERNING ENTITIES AND RELATIONSHIPS (LEVEL 1)

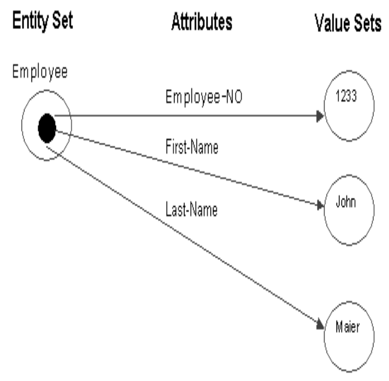


Level 1:	Entities	**Level 2:**	Entities/Relationship
	Entity sets		Relation
	Relationship Sets		
	Attributes		
	Values		
	Value Sets		
	Roles		

Levels of views of data.

Here we define an entity as a thing that can be distinctly identified. An example of an entity is a specific person, a company or an event. A relationship is an association among entities. Entities are classified into different entity sets. A relationship set is a mathematical relation among a number of entities, each taken from an entity set. The role of an entity is the function that it performs in the relationship.

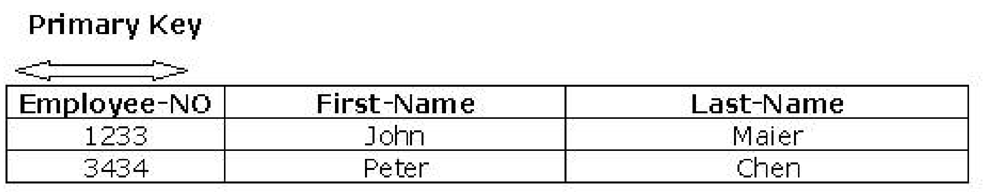
Values are classified into different value sets. An attribute can be formally defined as a function that maps from an entity set or a relationship set into a value set or a Cartesian product of value sets.



Attributes and values defined on an entity set.

3.3.2 INFORMATION STRUCTURE (LEVEL 2)

Entities, relationships and values, described at level 1, are conceptual objects in people's minds. At level 2 there is the consideration how to represent entities and relationships. Also primary keys are introduced to identify different entities in an entity set.



Primary Key

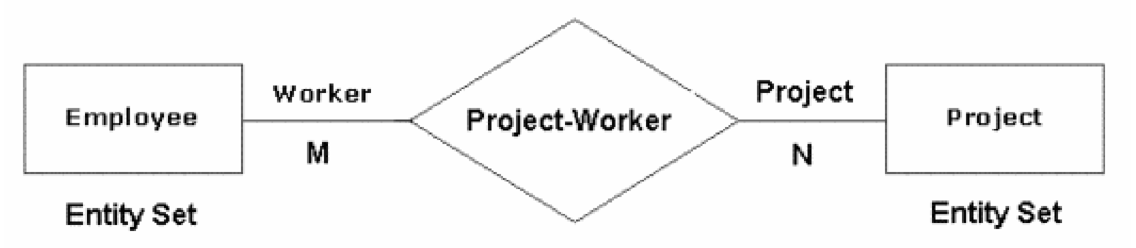
Employee-NO	First-Name	Last-Name
1233	John	Maier
3434	Peter	Chen

Entity relation in table form with a primary key.

Information about entities in an entity set and relationships can be organized in table form. The whole table is an entity relation and every row is an entity tuple.

3.4 ENTITY-RELATIONSHIP DIAGRAM

The entity relationship diagram is a diagrammatic technique to exhibit entities and relationships.

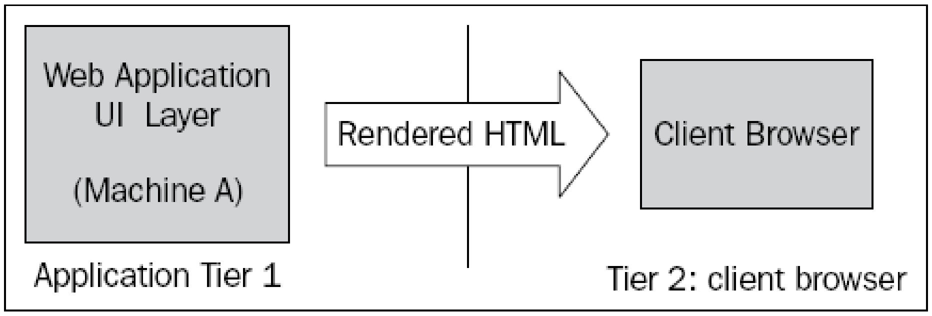


An entity relationship diagram .

Each entity is represented by a rectangular box and each relationship is represented by a diamond-shaped box. Lines, that connect the rectangular boxes, represent that the relationship set is defined on the connected rectangular boxes. The diagram is able to distinguish between 1:n, m:n and 1:1 mappings.

3.4.1 ER DIAGRAMS, DOMAIN MODEL, AND N-LAYER ARCHITECTURE WITH ASP.NET 3.5 (PART1)

Let us start with a 1-tier ASP.NET application configuration. *Note that the application as a whole including database and client browser is three tier.*



We can call this 1-tier architecture a 3-tier architecture if we include the browser and database (if used). For the rest of this unit we will ignore the database and browser as separate tiers so that we can focus on how to divide the main ASP.NET application layers logically, using the n-layer pattern to its best use.

We will first try to separate the data access and logical code into their own separate layers and see how we can introduce flexibility and re-usability into our solution. We will understand this with a sample project. Before we go ahead into the technical details and code, we will first learn about two important terms: ER Diagram and Domain Model, and how they help us in getting a good understanding of the application we need to develop.

Entity-Relationship diagrams, or ER diagrams in short, are graphical representations depicting relationships between different entities in a system. We humans understand and remember pictures or images more easily than textual information. When we first start to understand a project we need to see how different entities in the project relate to each other. ER diagrams help us achieve that goal by graphically describing the relationships.

> *An entity can be thought of as an object in a system that can be identified uniquely. An entity can have attributes; an attribute is simply a property we can associate with an entity. For example, a Car entity can have the following attributes: EngineCapacity, NumberofGears, SeatingCapacity, Mileage, and so on. So attributes are basically fields holding data to indentify an entity. Attributes cannot exist without an entity.*

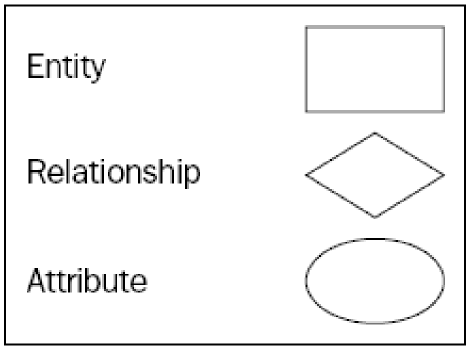
Let us understand ER diagrams in detail with a simple e-commerce example: a very basic Order Management System. We will be building a simple web based system to track customer's orders, and manage customers and products.

To start with, let us list the basic entities for our simplified Order Management System (OMS):

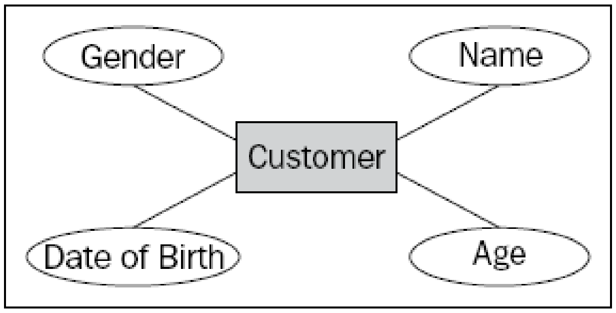
i. Customer: A person who can place Orders to buy Products.

ii. Order: An order placed by a Customer. There can be multiple Products bought by a Customer in one Order.
iii. Product: A Product is an object that can be purchased by a Customer.
iv. Category: Category of a Product. A Category can have multiple Products, and a Product can belong to many Categories. For example, a mixer-grinder can be under the Electronic Gadgets category as well as in Home Appliances.
v. OrderLineItem: An Order can be for multiple Products. Each individual Product in an order will be encapsulated by an OrderLineItem. So an Order can have multiple OrderLineItems.

Now, let us picture the relationship between the core business entities is defined using an Entity-Relationship diagram. Our ER diagram will show the relational associations between the entities from a database's perspective. So it is more of a relational model and will not show any of the object-oriented associations (for which we will use the Domain Model in the later sections of this unit). In an ER diagram, we show entities using rectangular boxes, the relationships between entities using diamond boxes and attributes using oval boxes, as shown below:

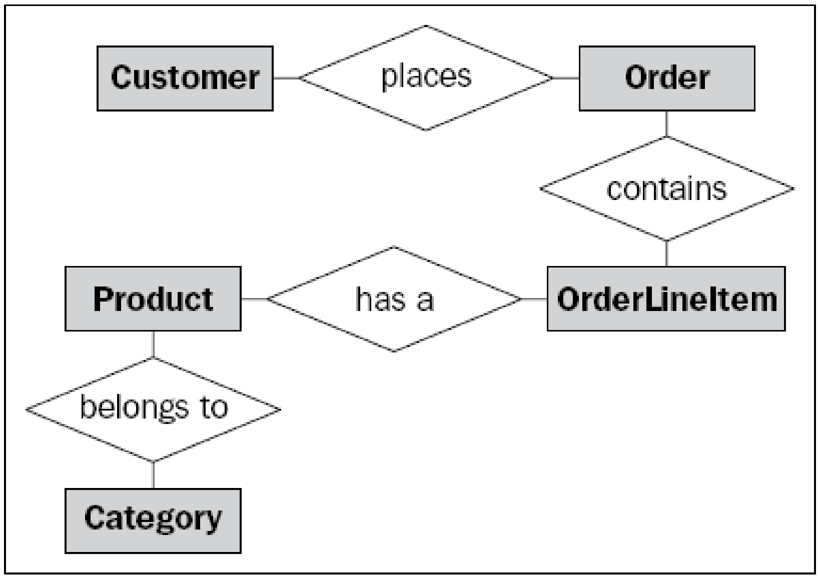


The purpose of using such shapes is to make the ER diagram clear and concise, depicting the relational model as closely as possible without using long sentences or text. So the Customer entity with some of the basic attributes can be depicted in an ER diagram as follows:



Now, let us create an ER diagram for our Order Management System. For the sake of simplicity, we will not list the attributes of the entities involved.

Here is how the ER diagram looks:



The above ER diagram depicts the relationships between the OMS entities but is still incomplete as the relationships do not show how the entities are quantitatively related to each other. We will now look at how to quantify relationships using degree and cardinality.

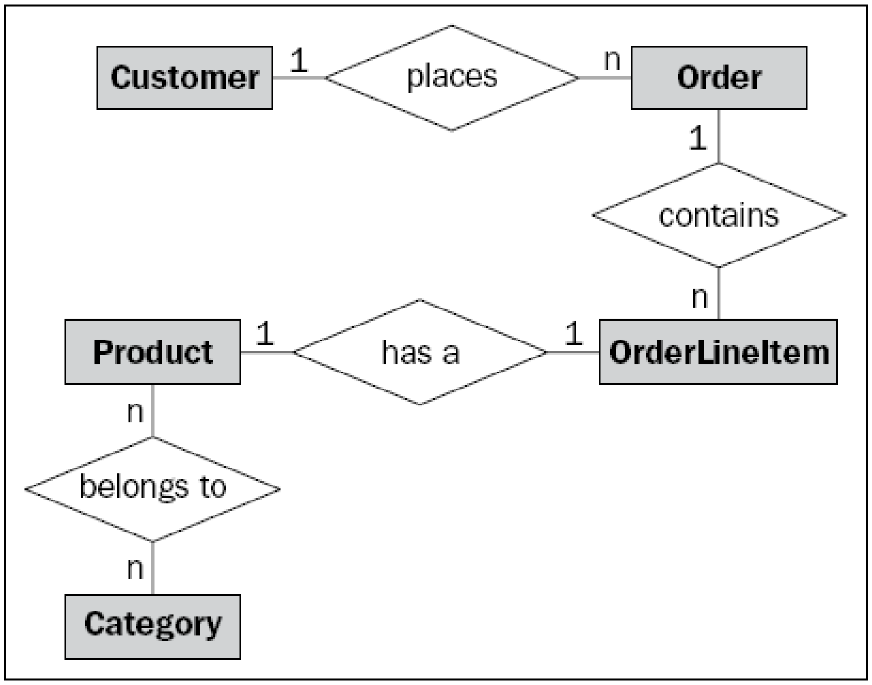
3.4.2 DEGREE AND CARDINALITY OF A RELATIONSHIP

The relationships in an ER diagram can also have a *degree*. A degree specifies the multiplicity of a relationship. In simpler terms, it refers to the number of entities involved in a relationship. All relationships in an OMS ER diagram have a degree of two, also called binary relationships. For example, in Customer-Order relationships only two entities are involved—Customer and Order; so it's a two degree relationship. Most relationships you come across would be binary.

Another term associated with a relationship is cardinality. The cardinality of a relationship identifies the number of instances of entities involved in that particular relationship. For example, an Order can have multiple OrderLineItems, which means the cardinality of the relationship between Order and OrderLineItem is one-to-many. The three commonly-used cardinalities of a relationship are:

i. One-to-one: Depicted as 1:1
   Example: One OrderLineItem can have only one Product; so the OrderLineItem and Product entities share a one-to-one relationship
ii. One-to-many: Depicted as 1:n
   Example: One customer can place multiple orders, so the Customer and Order entities share a one-to-many relationship
iii. Many-to-many: Depicted as n:m
   Example: One Product can be included in multiple Categories and one Category can contain multiple Products; therefore the Product and Category entities share a many-to-many relationship

After adding the cardinality of the relationships to our ER diagram, here is how it will look:



This basic ER diagrams tells us a lot about how the different entities in the system are related to each other, and can help new programmers to quickly understand the logic and the relationships of the system they are working on. Each entity will be a unique table in the database.

3.5 OMS PROJECT USING 2-LAYER

We know that the default coding style in ASP.NET 2.0 already supports the 1-tier 1-layer style, with two sub-layers in the main UI layer as follows:

i. Designer code files: ASPX markup files
ii. Code behind files: Files containing C# or VB.NET code

Because both of these layers contain the UI code, we can include them as a part of the UI layer. These two layers help us to separate the markup and the code from each other. However, it is still not advisable to have logical code, such as data access or business logic, directly in these code-behind files.

Now, one way to create an ASP.NET web application for our Order Management System (OMS) in just one layer is by using a DataSet (or DataReader) to fill the front-end UI elements directly in the code-behind classes. This will involve writing data access code in the UI layer (code-behind), and will tightly bind this UI layer with the data access logic, making the application rigid (inflexible), harder to maintain, and less scalable.

In order to have greater flexibility, and to keep the UI layer completely independent of the data access and business logic code, we need to put these elements in separate files. So we will now try and introduce some loose-coupling by following a 2-layer approach this time. What we will do is, write all data access code in separate class files instead of using the code-behind files of the UI layer. This will make the UI layer independent of the data-access code.

We are assuming that we do not have any specific business logic code at this point, or else we would have put that under another layer with its own namespace, making it a 3-layered architecture. We will examine this in the upcoming sections of this article.

3.5.1SAMPLE PROJECT

Let us see how we can move from this 1-tier 1-layer style to a 1-tier 2-layer style. Using the ER diagram above as reference, we can create a 2-Layer architecture for our OMS with these layers:

i. UI-layer with ASPX and code-behind classes
ii. Data access classes under a different namespace but in the same project

So let's start with a new VS 2008 project. We will create a new ASP.NET Web Project in C#, and add a new web form, *ProductList.aspx*, which will simply display a list of all the products using a Repeater control. The purpose of this project is to show how we can logically break up the UI layer further by separating the data access code into another class file.

The following is the ASPX markup of the ProductList page (unnecessary elements and tags have been removed to keep things simple):

```
<asp:Repeater ID="prodRepeater" runat="server">
     <ItemTemplate>
          Product Code: <%# Eval("Code")%>
              <br>
         Name: <%# Eval("Name")%>
              <br>
         Unit Price: $<%# Eval("UnitPrice")%>
              <br>
    </ItemTemplate>
</asp:Repeater>
```

In this ASPX file, we only have a Repeater control, which we will bind with the data in the code-behind file.

Here is the code in the *ProductList.aspx.cs* code-behind file:

```
namespace OMS
{
public partial class _Default : System.Web.UI.Page
    {
       /// <summary>
       /// Page Load method
       /// </summary>
       /// <param name="sender"></param>
       /// <param name="e"></param>
       protected void Page_Load(object sender, EventArgs e)
       {
           DataTable dt = DAL.GetAllProducts();
           prodRepeater.DataSource = dt;
           prodRepeater.DataBind();
       }
```

```
        }//end class
    }//end namespace
```

Note that we don't have any data access code in the code-behind sample above. We are just calling the *GetAllProducts()* method, which has all of data access code wrapped in a different class named DAL. We can logically separate out the code, by using different namespaces to achieve code re-use and greater architectural flexibility. So we created a new class named DAL under a different namespace from the UI layer code files. Here is the DAL code:

```
    namespace OMS.Code
    {
        public class DAL
        {
            /// <summary>
            /// Load all comments from the Access DB
            /// </summary>
            public static DataTable GetAllProducts()
            {
                string sCon =
    ConfigurationManager.ConnectionStrings[0].ConnectionString;
                using (SqlConnection cn = new SqlConnection(sCon))
                {
                    string sQuery = @"SELECT * FROM OMS_Product";
                    SqlCommand cmd = new SqlCommand(sQuery, cn);
                    SqlDataAdapter da = new SqlDataAdapter(cmd);
                    DataSet ds = new DataSet();
                    cn.Open();
                    da.Fill(ds);
                    return ds.Tables[0];
                }
            }
        }//end class
    }//end namespace
```

So we have separated the data access code in a new logical layer, using a separate namespace, *OMS.Code*, and using a new class. Now, if we want to, we can re-use the same code in the other pages as well. Furthermore, methods to add and edit a product can be defined in this class and then used in the UI layer. This allows multiple developers to work on the DAL and UI layers simultaneously.

Even though we have a logical separation of the code in this 2-layer sample architecture, we are still not using real Object Oriented Programming (OOP). All of the Object-Oriented Programming we have used so far has been the default structure the .NET framework has provided, such as the Page class, and so on.

When a project grows big in size as well as complexity, using the 2-layer model discussed above can become cumbersome and cause scalability and flexibility issues. If the project grows in complexity, then we will be putting all of the business logic code in either the DAL or the UI layer. This business logic code includes business rules. For example, if the customer orders a certain number of products in one order, he gets a certain level of discount. If we code such business rules in the UI layer, then if the rules change we need to change the UI as well, which is not ideal, especially in cases where we can have multiple UIs for the same code, for example one normal web browser UI and another mobile-based UI.

We also cannot put business logic code in the DAL layer because the DAL layer should only contain data access code which should not be mixed with any kind of business processing logic. In fact the DAL layer should be quite "dumb"–there should be no "logic" inside it because it is mostly a utility layer which only needs to put data in and pull data out from a data store.

To make our applications more scalable and to reap the benefit of OOP, we need to create objects, and wrap business behavior in their methods. This is where the Domain Model comes into the picture.

3.6 DOMAIN MODEL USING UML

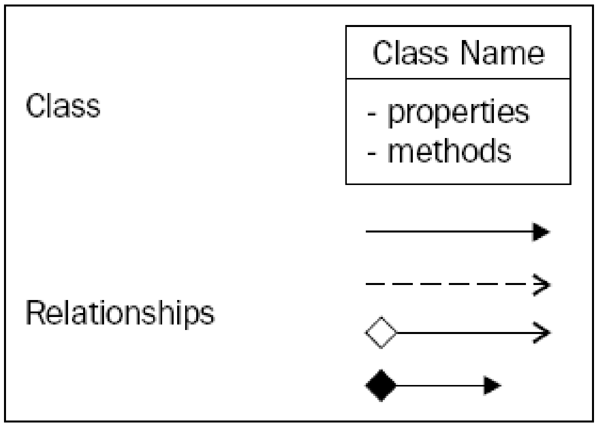
The domain model is a more object-oriented way of indicating the relationships between different objects in the context of the business logic of the application. It is similar to the ER diagram. But instead of merely showing the relationships between the entities involved, it graphically reflects how these entities relate to each other in an object-oriented fashion. On the other hand, an ER diagram is only focused from a relational perspective.

Unified Modeling Language, or UML in short, is a graphical language used to describe object-oriented designs for software systems. UML is quite a vast language, but we will focus more on class diagrams and UML relationships to represent our domain model. Class diagrams are widely used in every object-oriented system to describe the different types of internal relationships between the different business entities.

Before going for a 3-layer object-oriented system, we need to create a domain model of the system. So we need to put all of the business code into separate logical structures and start creating a domain model, in order to understand the different business entities involved.

For this, we need to "organize" the code by breaking it down into logical entities, which we call objects, and create relationships between them. The resulting set of objects with relationships defined between them would be known as the domain model of the application. It is so called because this model illustrates how different entities in the application domain would interact with each other.

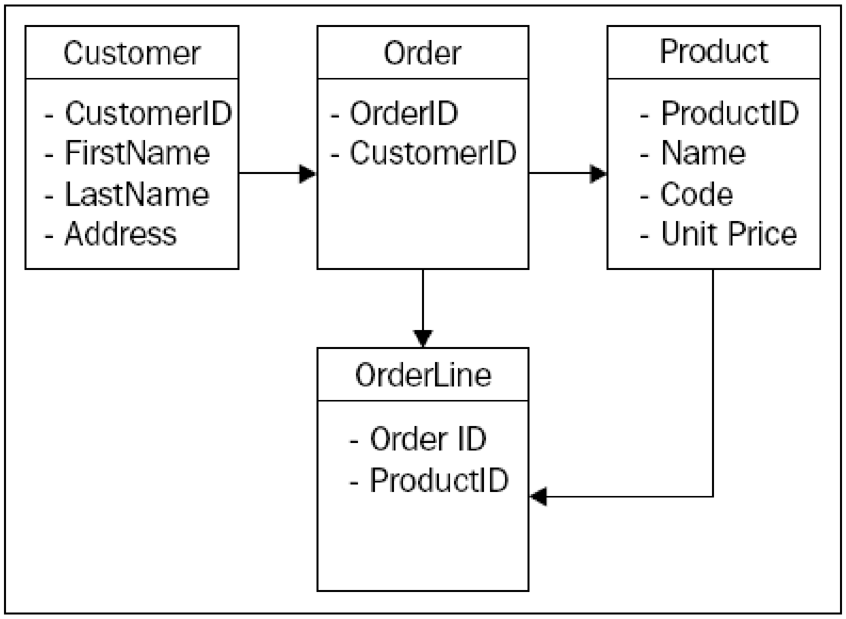
We use the following shapes in our class diagram:



We will learn in detail what each figure represents, and how to create a domain model using them.

3.6.1 CLASS DIAGRAM

A class diagram simply represents how different entities are related to each other in an object-oriented system. Class diagrams are different from ER diagrams because class diagrams deal with relationships in an object-oriented manner, showing inheritance, interfaces and so on, whereas an ER diagram can only depict relational models (for Relational Database Management Systems, or RDBMSs). In order to create a class diagram for our OMS, let us highlight the major entities in our OMS in terms of domain classes:



The rectangular boxes denote the entities (or classes) with the class name in the header and the attributes (or fields) below it. The arrows define relationships between entities. These relationships can be of different types and are depicted differently using different arrow styles.

Broadly speaking, we can place class relationships into these categories:

i. Dependency relationship
ii. Association
iii. Generalization
iv. Realization

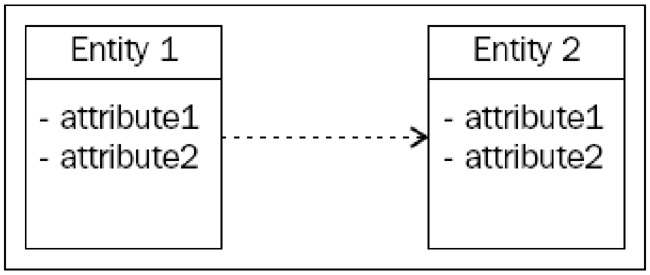
Let's explore each of these UML relationships in detail.

3.7 UML Relationships

In an ER diagram for two entities A and B, we can show only one type of relationship–a Relational relationship–which means that entity A is somehow related to entity B. But in a class diagram, the relationships can be further divided on the basis of object-oriented principles such as inheritance, association, and so on. The following sections describe the main UML relationships used in a class diagram.

3.7.1DEPENDENCY RELATIONSHIP

A **Dependency** exists between two elements if changes to one element will affect the other. A dependency relationship is the simplest relationship of all, and means that Entity 1 depends on Entity 2 in such a way that any change in entity 2 might break entity 1. This is a one-way relationship only—changes in entity 1 will not affect entity 2 in any manner. Dependency relationships are represented by a broken (dashed) line with an "empty" arrow (--->). The direction of this arrow flows to the entity that is dependent on the entity that the arrow flows from.



A simple example would be:

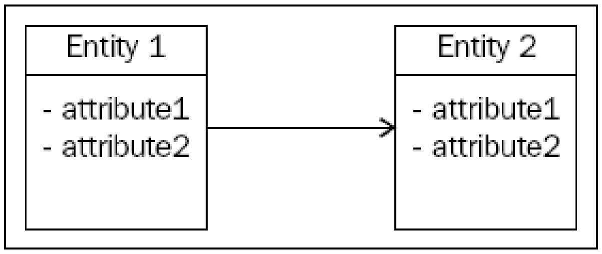
```
public class Entity1
{
    public void MethodX (Entity2 en)
    {
        // . . .
    }
    public void MethodY ()
    {
        Entity2 en2 = new Entity2();
        // . . .
    }
}
```

In the above pseudo code, Entity2 is used in Entity1 first as a method parameter and secondly inside *MethodY()* as a local variable. Both of these cases show a dependency relationship.

An important point about dependency relationships relates to the state of the objects involved. The state of Entity2 is not related to the state of Entity-1, as Entity2 is not a member of the Entity1 class. It is only used in the methods as local variable.

3.7.2 Association Relationship

An **Association** shows the relationship between instances of classes. An association is similar to a dependency except that it specifies the relationship in a stricter form. An association means that instead of Entity2 being used in Entity1 as a local variable, it would be a global variable instead, which can be a private, public, or protected class member. In its basic form, it is represented by a solid arrow (instead of dashed as in a dependency relationship).

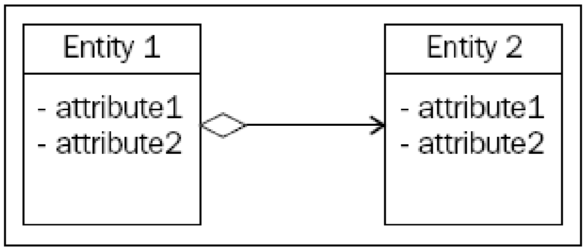


```
public class Order //entity1
{
  private Customer _customer;
  public Customer GetCustomer ()
  {
    return _customer;
  }
}
```

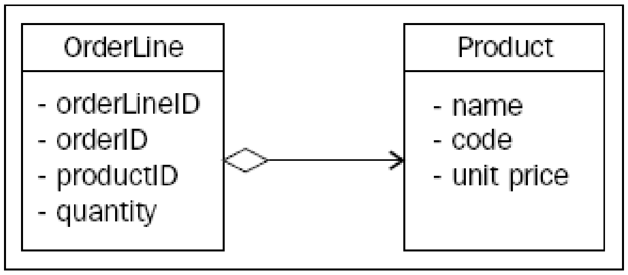
In the above pseudo code, the Customer object is a part of the Order class—a private member in this case. This means that the Customer object forms a part of the state of the Order. If you have an Order object, you can easily identify the Customer associated with it. This is not possible in a dependency. Hence, the association is a stronger form of dependency. An Association relationship can be divided further into two separate relationships, based on the state of the aggregated object in the dependent class.

iv. Aggregation

An **Aggregation** relationship depicts a classifier as a part of, or as subordinate to, another classifier. For example, if Entity1 goes out of scope, it does not mean that Entity2 has to go out of scope too. That is, the lifetime of Entity2 is not necessarily controlled by Entity1. An aggregation is represented by a straight arrow, with an empty diamond at the tail, as shown in the following figure:



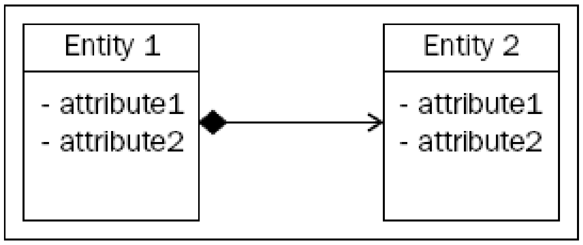
So, in our example, Entity2 is a part of (or subordinate to) Entity 1. If you destroy the parent class (Entity 1) in an aggregation (weak) relationship, the child class (Entity 2) can survive on its own. Let's understand aggregations by using our example of the Order Management System. Consider the OrderLine and Product classes. An OrderLine can have multiple quantities of one Product. If an OrderLine is destroyed, it does not mean that we delete the Product as well. A Product can exist independently of the OrderLine object. Here is the relationship diagram between OrderLine and Product classes:



In the diagram, we can see an Aggregation relationship between OrderLine and Product classes. Put simply, the above diagram states that if an order is cancelled, all of the products will not be destroyed; they will only be "de-associated" from that particular order.

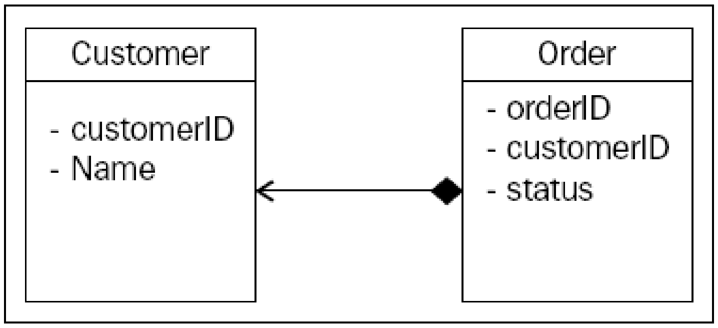
v. Composition

A **Composition** is exactly like Aggregation except that the lifetime of the 'part' is controlled by the 'whole'. For example: You have a 'student' who has a 'schedule'. If you destroy the student, the schedule will cease to exist. In this case, the associated entity is destroyed when the parent entity goes out of scope. Composition is represented by a straight arrow with a solid diamond at the tail, as shown below.



In our case, Entity-2 is controlled by Entity-1. If Entity 1 is destroyed in a composition (strong) relationship, Entity-2 is destroyed as well.

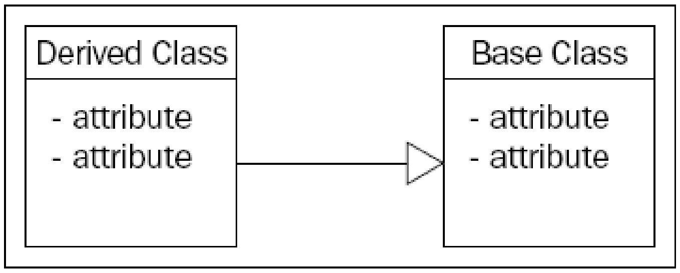
Let's understand compositions by using our example of the Order Management System. Consider the Customer and Order classes. A Customer can have one or more orders, and an Order can have one or more Products (in order lines). An Order object cannot exist on its own without a Customer. So the following Composition indicates that if a Customer object goes out of scope, the Orders associated with that Customer go out of scope too.



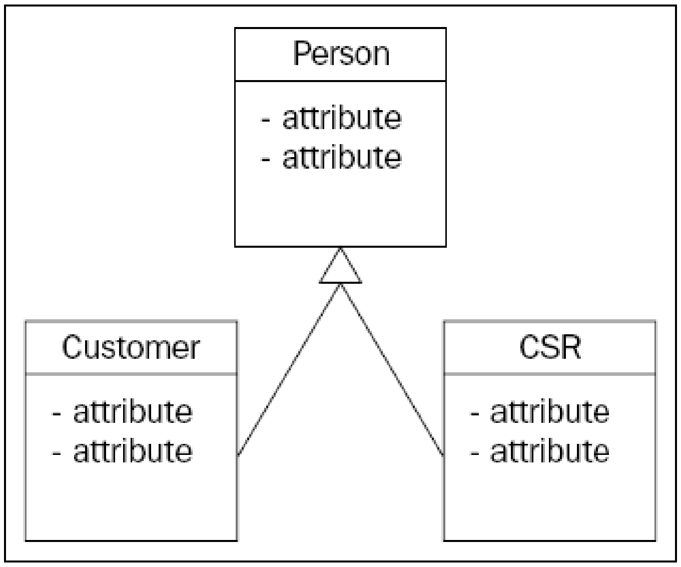
3.7.3 GENERALIZATION RELATIONSHIP

Inheritance is a very widely known and common feature of OOP. In UML, inheritance is depicted using generalization relationships, depicted by a straight arrow with a hollow arrowhead (triangle) at one end. A generalization relationship (also known as a "is-a" relationship) implies that a specialized (child) class is based on a general (parent) class.

Here is a diagram illustrating this:



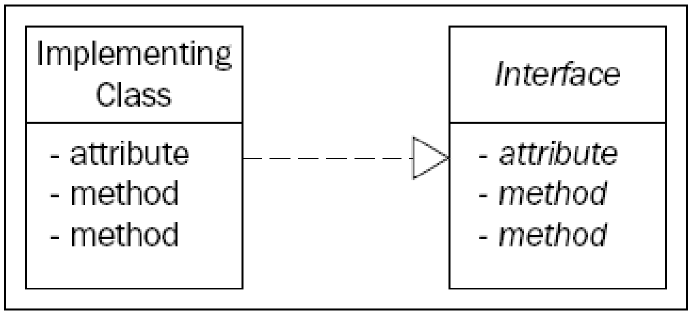
Here, we can see that the arrow points in the direction of the base class. In our Order Management System, we can have a base class for all customers; we can call it Person class so that we have other classes derived from it, such as Customer, CSR (Customer Sales Representative), and so on.



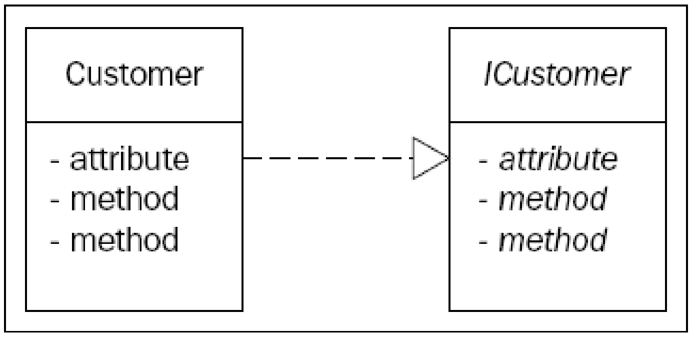
3.7.4 REALIZATION RELATIONSHIP

Realization is similar to generalization but depicts the relationship between an interface and a class implementing that interface. In UML, realization is depicted with a dashed arrow with a hollow arrowhead (triangle) at one end. A realization relationship exists between the two classes when one of them must realize, or implement, the behavior specified by the other.

For example, a realization relationship connects an interface to an implementing class. The interface specifies the behaviors, and the implementing class implements the behaviors. Here is a diagram illustrating this:



Here, we can see that the arrow points in the direction of the interface. Note that the italicized text in entities that are interfaces. It is UML convention to italicize interfaces.



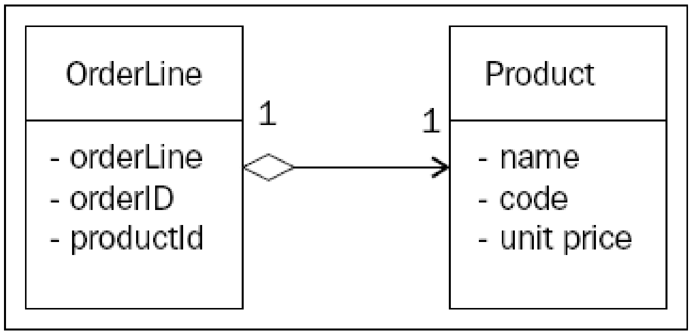
Activity B/self assessment exercise

i. **What are the similarities in realization and generalization**
ii. **Describe the concepts of association and dependency relationships**

3.8 MULTIPLICITY

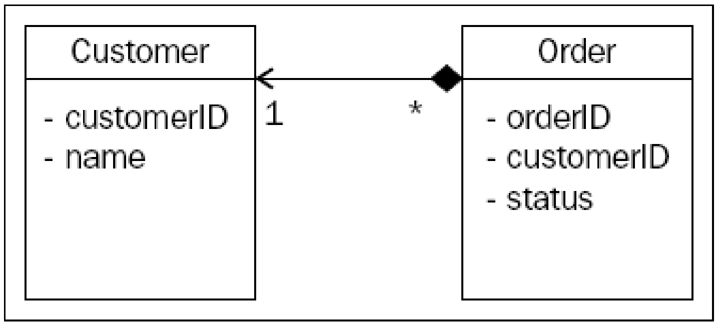
Multiplicity quantifies the relationship between two entities. Multiplicity is closely related to the cardinality of a relationship, which we learned about earlier when discussing ER diagram. Multiplicity indicates how many instances of classes (objects) are related to each other in a UML relationship. The following is a list of different multiplicities we can have between two entities in a class diagram:

iii. One-to-one: For example, one OrderLine object can have only one product. This is depicted as follows:



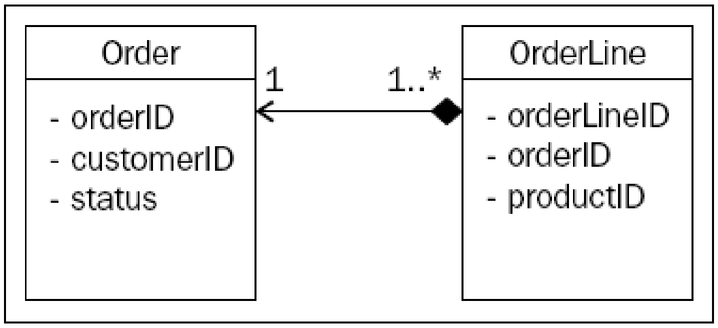
Note how we show a 1:1 multiplicity using the number "1" at the end points of the aggregation relationship.

iv. One-to-many: For example, a customer can have many orders. This is depicted as follows:



Note the text "1" and "*" near the entities; these are multiplicity indicators. In the above example, the multiplicity indicates that one (1) customer can have multiple orders (*). We depict "many" using a "*" (asterisk) symbol.

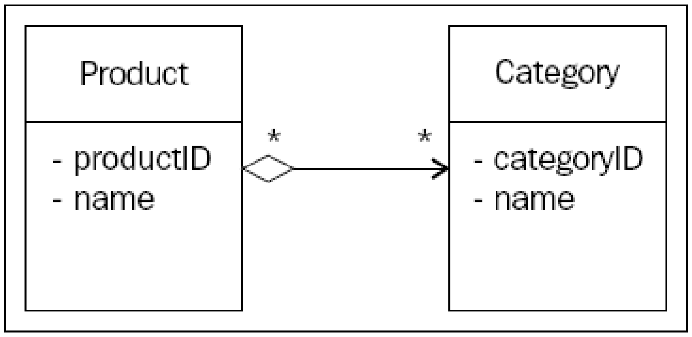
The relationship between Order and OrderLine objects is the same. An order can have multiple products; each product will be shown in a separate line (called as OrderLine) in the Order. So there can be one or more order lines for a single order, as shown here:



The above diagram confirms that for each order, there will be one or more order lines. We can't use 0..* here in place of 1..* because each order will have atleast one product in it (as one order line item).

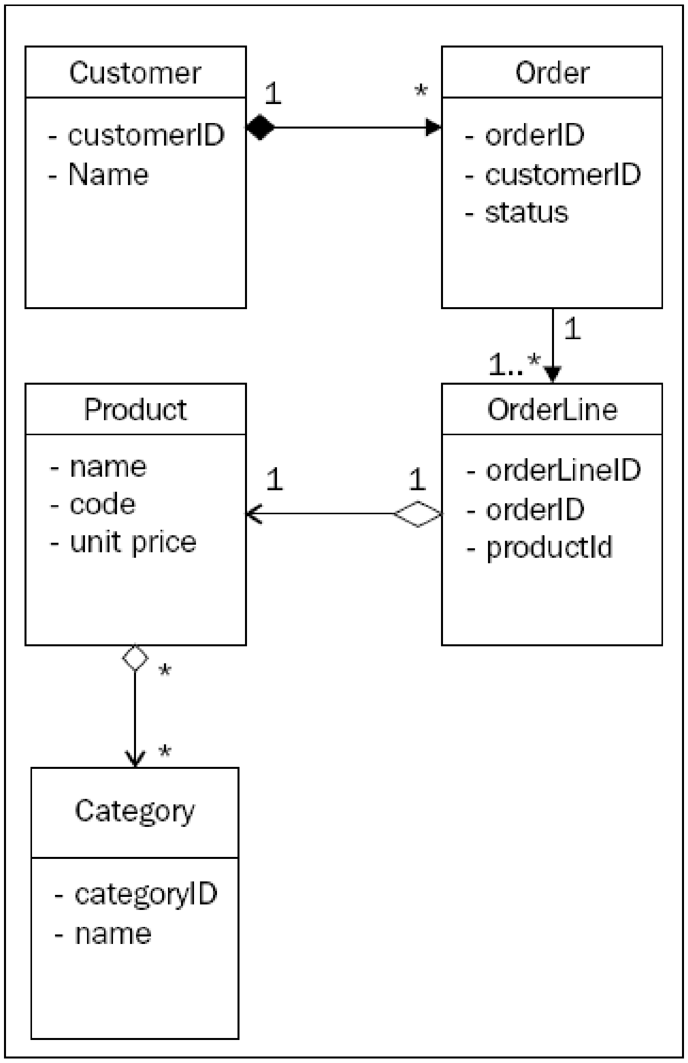
Also, if an order gets cancelled (destroyed), then all order lines will be destroyed. It doesn't make sense to have order lines that are not a part of any order—hence the composition.

v. Many-to-many: A Product can belong to multiple Categories, and a Category object can include multiple Product objects. To depict such many-to-many relationships, we use asterisk at both ends of the relationship arrow, as shown here:



Also note the aggregation relationship between the Product and the Category, because both can exist independently of each other.

So, now, we can combine all of the above diagrams and create a simple class diagram with all of the relationships and multiplicities for our OMS. Here is the combined UML class diagram for our sample application:



So we have a very simple domain model of a simple Order Management System. Now, based on the above classes, let's look at how we can convert this domain model to code by creating a 1-tier 3-layer architecture based web application.

Tutor Marked Assessment

i. study the order management system and re-implement t using a different OOP language

ii. with the aid of a diagram describe the concept of multiplicity

4.0 CONCLUSION AND SUMMARY

All the samples we covered in this unit were of the 1-tier n-layer style. We learned how to create a 1-tier 2 layer architecture using logical code separation. Then we focused on the need for a 3-layered solution and examined ER-diagrams, domain models and UML, all of which are important tools that aid in the understanding of commercial projects required to build a 3-layered structure.

Reference

1. Chen, P.P., "The Entity-Relationship Model: Toward a Unified View of Data," ACM Trans. on Database Systems, Vol.1, No.1, March 1976, pp. 9-36.
2. Chen, Peter P., Entity-Relationship Modelling: Historical Events, Future Trends, and Lessons Learned, Springer-Verlag New York, Inc, 2002, pp. 296-310.
3. Winslett, Marianne : Peter Chen Speaks Out on Paths to Fame, the Roots of ER Model in Human Language, the ER Model in Software Engineering, the Need for ER Databases and More, SIGMOD Record, Vol. 33, No. 1, March 2004
4. http://www.packtpub.com/application-architecture-and-design-for-asp-.net-3.5/book
5. A.P.G. Brown, Modelling a Real-World System and Designing a Schema to Represent It, in Data Base Description, ed Douque and Nijssen, North-Holland, 1975,
6. Beynon-Davies (2004). Database Systems. Houndmills, Basingstoke, UK: Palgrave
7. John Currier. "SchemaSpy: Graphical Database Schema Metadata Browser".
8. Richard Barker (1990). *CASE Method: Tasks and Deliverables*. Wokingham, England: Addison-Wesley.
9. Paul Beynon-Davies (2004). *Database Systems*. Houndmills, Basingstoke, UK: Palgrave
10. http://en.wikipedia.org/wiki/Entity-relationship_model

Further Reading

1. An Entity Relationship Diagram Example. Demonstrates the crow's feet notation by way of an example.
2. "Entity-Relationship Modeling: Historical Events, Future Trends, and Lessons Learned" by Peter Chen.
3. "English, Chinese and ER diagrams" by Peter Chen.
4. Case study: E-R diagram for Acme Fashion Supplies by Mark H. Ridley.
5. Logical Data Structures (LDSs) - Getting started by Tony Drewry.
6. Introduction to Data Modeling

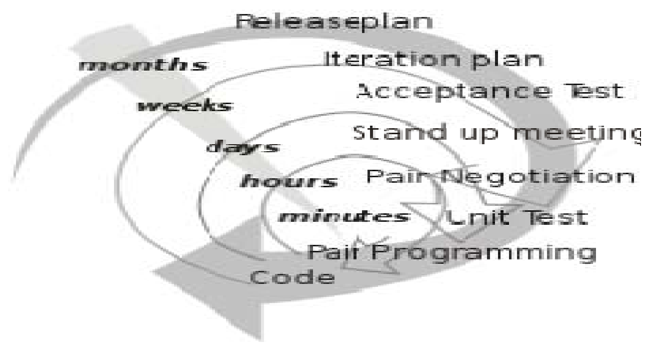
Contents

1.0 Introduction

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 short development cycles (timeboxing), which is intended to improve productivity and introduce 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. It is unrelated to "cowboy coding", which is more free-form and unplanned. It does not advocate "death march" work schedules, but instead working at a sustainable pace.

Critics have noted several potential drawbacks, including problems with unstable requirements, no documented compromises of user conflicts, and lack of an overall design spec or document.



Planning and feedback loops in Extreme Programming (XP) with the time frames of the multiple loops.

2.0 Objectives

By the end of this unit you are expect to be able to:

i. have acquired the skills necessary to improve software quality to meet the customer needs
ii. be able to put in place checkpoints to ascertain where job is in tandem with the initial plan
iii. have a good mastery of the extreme programming elements
iv. differentiate XP from others methodologies

3.0 History

Extreme Programming was created by Kent Beck during his work on the Chrysler Comprehensive Compensation System (C3) payroll project. Beck became the C3 project leader in March 1996 and began to refine the development method used in the project and wrote a book on the method (in October 1999, *Extreme Programming Explained* was published). Chrysler cancelled the C3 project in February 2000, after the company was acquired by Daimler-Benz.

Although Extreme Programming itself is relatively new, many of its practices have been around for some time; the methodology, after all, takes "best practices" to extreme levels. For example, the "practice of test-first development, planning and writing tests before each micro-increment" was used as early as NASA's Project Mercury, in the early 1960s (Larman 2003). Refactoring, modularity, bottom-up and incremental design were described by Leo Brodie in his book published in 1984

3.2.1 Origins

Most software development in the 1990s was shaped by two major influences: internally, object-oriented programming replaced procedural programming as the programming paradigm favoured by some in the industry; externally, the rise of the Internet and the dot-com boom emphasized speed-to-market and company-growth as competitive business factors. Rapidly-changing requirements demanded shorter product life-cycles, and were often incompatible with traditional methods of software development.

The Chrysler Comprehensive Compensation System was started in order to determine the best way to use object technologies, using the payroll systems at Chrysler as the object of research, with Smalltalk as the language and GemStone as the data access layer. They brought in Kent Beck, a prominent Smalltalk practitioner, to do performance tuning on the system, but his role expanded as he noted several issues they were having with their development process. He took this opportunity to propose and implement some changes in their practices based on his work with his frequent collaborator, Ward Cunningham.

Beck invited Ron Jeffries to the project to help develop and refine these methods. Jeffries thereafter acted as a coach to instill the practices as habits in the C3 team.

Information about the principles and practices behind XP was disseminated to the wider world through discussions on the original Wiki, Cunningham's WikiWikiWeb. Various contributors discussed and expanded upon the ideas, and some spin-off methodologies resulted.

3.2.2 Current state

XP created quite a buzz in the late 1990s and early 2000s, seeing adoption in a number of environments radically different from its origins.

The high discipline required by the original practices often went by the wayside, causing some of these practices that were thought too rigid to be deprecated or left undone on individual sites. Agile development practices have not stood still, and XP is still evolving, assimilating more lessons from experiences in the field. In the second edition of *Extreme Programming Explained*, Beck added more values and practices and differentiated between primary and corollary practices.

3.3 CONCEPT

3.3.1 GOALS

Extreme Programming Explained describes Extreme Programming as a software development discipline that organizes people to produce higher quality software more productively.

In traditional system development methods (such as 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 doctrine 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.

Activity A

i. describe what you understand by extreme programming
ii. describe the concept of "if some is good more is better" as applies to XP
iii. mention some notable drawbacks in XP
iv. differentiate between XP and waterfall model

3.3.2 Activities

XP describes four basic activities that are performed within the software development process.

i. Coding

The advocates of XP argue that the only truly important product of the system development process is code - software instructions a computer can interpret. 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 problem, one should simply code all solutions and determine with automated tests which solution is most suitable. 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.

ii. 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 successfully, 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.

A "testathon" is an event when programmers meet to do collaborative test writing, a kind of brainstorming relative to software testing.

iii. Listening

Programmers must listen to what the customers need the system to do, what "business logic" is needed. They must understand these needs well enough to give the customer feedback about the technical aspects of how the problem might be solved, or cannot be solved. Understanding of his or her problem would assist in creating software that would solve the customers needs and make room for business expansion. This Communication between the customer and programmer would make programmer's job easier and provide greater customer satisfaction.

iv. 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 a long way without designing but at a given time one will get stuck. The system becomes too complex and the dependencies within the system cease to be clear. One can avoid this by creating a design structure that organizes the logic in the system. Good design will avoid lots of dependencies within a system; this means that changing one part of the system will not affect other parts of the system.

3.4. Values

Extreme Programming initially recognized four values in 1999. A new value was added in the second edition of *Extreme Programming Explained*. The five values are:

i. Communication

Building software systems requires communicating system requirements to the developers of the system. In formal software development methodologies, this task is accomplished through documentation. Extreme Programming techniques can be viewed as methods for rapidly building and disseminating 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 favours simple designs, common metaphors, collaboration of users and programmers, frequent verbal communication, and feedback.

ii. Simplicity

Extreme Programming encourages starting with the simplest solution. Extra functionality can then be added later. The difference between this approach and more conventional 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 gonna 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 future 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.

iii. Feedback

Within Extreme Programming, feedback relates to different dimensions of the system development:

a. 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.

b. Feedback from the customer: The functional tests (aka 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.
c. 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*. To quote Kent Beck, "Optimism is an occupational hazard of programming, feedback is the treatment."

i. 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 implement anything else. Courage enables developers to feel comfortable with refactoring their code when necessary. This means reviewing the existing system and modifying it so that future changes can be implemented 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.

ii. Respect

The respect value manifests 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 at hand through refactoring.
Adopting the four earlier values leads to respect gained from others in the team. Nobody on the team should feel unappreciated or ignored. This ensures 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.

3.4.2 Rules

The first version of XP rules was proposed by Ken Hauer in XP/Agile Universe 2003. He felt XP was defined by its rules, not its practices (which are subject to more variation and ambiguity). He defined two categories: "Rules of Engagement" which dictate the environment in which software development can take place effectively, and "Rules of Play" which define the minute-by-minute activities and rules within the framework of the Rules of Engagement.

In the APSO workshop at ICSE 2008 Conference, Mehdi Mirakhorli proposed a new and more precise and comprehensive version of the Extreme Programming Rules, more independent of the practices, and in intended to be more "agile".

3.4.1 Rules of engagement

According to Mehdi Mirakhorli, these are:

i. Business people and developers do joint work: Business people and developers must work together daily throughout the project.
ii. Our highest priority is customer satisfaction: The customer must set and continuously adjust the objectives and priorities based on estimates and other information provided by the developers or other members of the team. Objectives are defined in terms of what not how.
iii. Deliver working software frequently: Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter time scale (timeboxing).
iv. Working software: Working software is the primary measure of progress.
v. Global awareness: At any point, any member of the team must be able to measure the team's progress towards the customer's objectives and the team reflects on how to become more effective, then tunes and adjusts its behaviour accordingly.
vi. The team must act as an effective social network, which means:
    a. Honest communication leading to continuous learning and an emphasis on person-to-person interaction, rather than documentation.
    b. Minimal degrees of separation from what is needed by the team to make progress and the people/resources that can meet those needs.
    c. Alignment of authority and responsibility.

Activity B

i. Briefly explain how coding can be used to determine the best solution to a problem in XP
ii. Describe the essence of communication in XP
iii. Explain in details the principles upon which XP is built

3.5 Principles

The principles that form the basis of XP are based on the values just described and are intended to foster decisions in a system development project. The principles are intended to be more concrete than the values and more easily translated to guidance in a practical situation.

i. Feedback

   Extreme Programming sees feedback as most useful if it is done rapidly and expresses that the time between an action and its feedback is critical to learning and making changes. Unlike traditional system development methods, contact with the customer occurs in more frequent iterations. The customer has clear insight into the system that is being developed. He or she can give feedback and steer the development as needed. Unit tests also contribute to the rapid feedback principle. When writing code, the unit test provides direct feedback as to how the system reacts to the changes one has made. If, for instance, the changes affect a part of the system that is not in the scope of the programmer who made them, that programmer will not notice the flaw. There is a large chance that this bug will appear when the system is in production.

ii. Assuming simplicity

This is about treating every problem as if its solution were "extremely simple". Traditional system development methods say to plan for the future and to code for reusability. Extreme programming rejects these ideas.
The advocates of Extreme Programming say that making big changes all at once does not work. Extreme Programming applies incremental changes: for example, a system might have small releases every three weeks. When many little steps are made, the customer has more control over the development process and the system that is being developed.

iii. Embracing change

The principle of embracing change is about not working against changes but embracing them. For instance, if at one of the iterative meetings it appears that the customer's requirements have changed dramatically, programmers are to embrace this and plan the new requirements for the next iteration.

3.6 Practices

Extreme Programming has been described as having 12 practices, grouped into four areas:

Fine scale feedback

i. Pair programming
ii. Planning game
iii. Test-driven development
iv. Whole team

Continuous process

i. Continuous integration
ii. Refactoring or design improvement
iii. Small releases

Shared understanding

i. Coding standards
ii. Collective code ownership
iii. Simple design
iv. System metaphor

Programmer welfare

Sustainable pace

Coding

i. The customer is always available
ii. Code the Unit test first

iii. Only one pair integrates code at a time
iv. Leave Optimization till last
v. No Overtime

Testing

i. All code must have Unit tests
ii. All code must pass all Unit tests before it can be released.
iii. When a Bug is found tests are created before the bug is addressed (a bug is not an error in logic, it is a test you forgot to write)
iv. Acceptance tests are run often and the results are published

4.0 Conclusion

Extreme Programming encourages feedback done rapidly and expresses that the time between an action and its feedback is critical to learning and making changes. It tends to treat every problem as if its solution were extremely simple.

Tutor Marked Assessment Test

i. list and explain ten different things you will consider in a software acceptance test

5.0 Summary

i. Extreme Programming has 12 practices grouped into four areas:

a. Fine scale feedback b.

Programmer welfare c.

Shared understanding d.

Continuous process

ii. The principles that form the basis of XP are based on the values and are intended to foster decisions in a system development project.

iii.XP describes four basic activities that are performed within the software development process such as coding, testing, listening and designing.

References

1. "Human Centred Technology Workshop 2005", 2005, PDF webpage: Informatics-UK-report-cdrp585.
2. Design Patterns and Refactoring", University of Pennsylvania, 2003, webpage: UPenn-Lectures-design-patterns.
3. "Extreme Programming" (lecture paper), USFCA.edu, webpage: USFCA-edu-601-lecture.
4. "Manifesto for Agile Software Development", Agile Alliance, 2001, webpage: Manifesto-for-Agile-Software-Dev
5. Everyone's a Programmer" by Clair Tristram. *Technology Review*, Nov 2003. p. 39

6. Extreme Programming", *Computerworld* (online), December 2001, webpage: Computerworld-appdev-92.
7. *Extreme Programming Refactored: The Case Against XP*. ISBN 1590590961.
8. Brodie, Leo (2008) (paperback). Thinking Forth. Prentice-Hall. ISBN 0-13-917568-7. http://thinking-forth.sourceforge.net/.
9. Ken Auer The Case Against Extreme Programming: A Self-Referential Safety Net
10. Cutter Consortium : Industrial XP: Making XP Work in Large Organizations Extreme Programming (XP) Six Sigma CMMI

Contents

1.0 Introduction

You can best achieve requirements success by applying established good practices on your projects. Thoughtfully tailor the practices to suit your project type, constraints, and organizational culture. Some highly exploratory or innovative projects can tolerate the excessive rework that results from informal requirements engineering. Most development efforts will benefit from a more deliberate and structured approach, though. Telepathy and clairvoyance rarely suffice.

2.0 Objective

By the end of this unit the student is expected to understand the following:

i. What requirement engineering is

ii. Types and categories of requirement engineering

iii. How to design business needs document

iv. The types, processes and requirements of requirement engineering

3.0 Definition Requirements Engineering

Requirement engineering is a sub discipline of systems engineering and software engineering that is concerned with determining the goals, functions, and constraints of hardware and software systems. In some life cycle models, the requirement engineering process begins with a feasibility study activity, which leads to a feasibility report. If the feasibility study suggests that the product should be developed, then requirement analysis can begin. If requirement analysis precedes feasibility studies, which may foster outside the box thinking, then feasibility should be determined before requirements are finalized. A **requirement** is defined as "a condition or capability to which a system must conform".

Requirement engineering is sometimes referred to loosely by names such as *requirements gathering*, *requirements capture*, or *requirements specification*. The term *requirements analysis* can also be applied specifically to the analysis proper, as opposed to elicitation or documentation of the requirements, for instance.

System requirements should specify *what*, not *how*. Why? Solution should not be pre-defined. Requirements should allow for a wide range of proposed alternative solutions. It is defined as (requirements) Customer generated, natural language document. It enumerates system services and constraints. The software industry is exhibiting an increasing interest in requirements engineering that is, understanding what you intend to build before you're done building it.

3.1 Categories Of Requirement Engineering

There are many different kinds of requirements. One way of categorizing them is described as the **FURPS+** model, using the acronym FURPS to describe the major categories of requirements with subcategories as shown below.

i. Functionality
ii. Usability
iii. Reliability
iv. Performance
v. Supportability

The "+" in FURPS+ reminds you to include such requirements as:

i. Design constraints
ii. Implementation requirements
iii. Interface requirements
iv. Physical requirements.

3.2 Functional requirements

Specify actions that a system must be able to perform, without taking physical constraints into consideration. These are often best described in a use-case model and in use cases. Functional requirements thus specify the input and output behavior of a system.

Requirements that are not functional, such as the ones listed below, are sometimes called **non-functional requirements**. Many requirements are non-functional, and describe only attributes of the system or attributes of the system environment. Although some of these may be captured in use cases, those that cannot may be specified in Supplementary Specifications. Nonfunctional requirements are those that address issues such as those described below.

A complete definition of the software requirements, use cases, and Supplementary Specifications may be packaged together to define a **Software Requirements Specification (SRS)** for a particular "feature" or other subsystem grouping.

3.2.1 Functionality

Functional requirements may include:

i. Feature Sets
ii. Capabilities
iii. Security

3.2.2 Usability

Usability requirements may include such subcategories as:

i. Human factors (see Concepts: User-Centered Design)
ii. Aesthetics
iii. Consistency in the user interface (see Guidelines: User-Interface)
iv. Online and context-sensitive help
v. Wizards and agents
vi. User documentation
vii. Training materials

3.2.3 Reliability

Reliability requirements to be considered are:

i. Frequency And Severity Of Failure
ii. Recoverability
iii. Predictability
iv. Accuracy
v. Mean Time Between Failure (Mtbf)

3.2.4 Performance

A performance requirement imposes conditions on functional requirements. For example, for a given action, it may specify performance parameters for:

i. Speed
ii. Efficiency
iii. Availability
iv. Accuracy
v. Throughput
vi. Response Time
vii. Recovery Time
viii. Resource Usage

3.2.5 Supportability

Supportability requirements may include:

i. Testability
ii. Extensibility
iii. Adaptability

iv. Maintainability
v. Compatibility
vi. Configurability
vii. Serviceability
viii. Installability
ix. Localizability (Internationalization)

3.2.6 Design Requirement

A design requirement, often called a **design constraint**, specifies or constrains the design of a system.

3.2.7 Implementation Requirement

An implementation requirement specifies or constrains the coding or construction of a system. Examples are:

i. Required Standards
ii. Implementation Languages
iii. Policies For Database Integrity
iv. Resource Limits
v. Operation Environments

3.2.8 Interface Requirement

An interface requirement specifies:

i. An external item with which a system must interact
ii. Constraints on formats, timings, or other factors used by such an interaction

3.2.9 Physical Requirement

A physical requirement specifies a physical characteristic that a system must possess; for example,

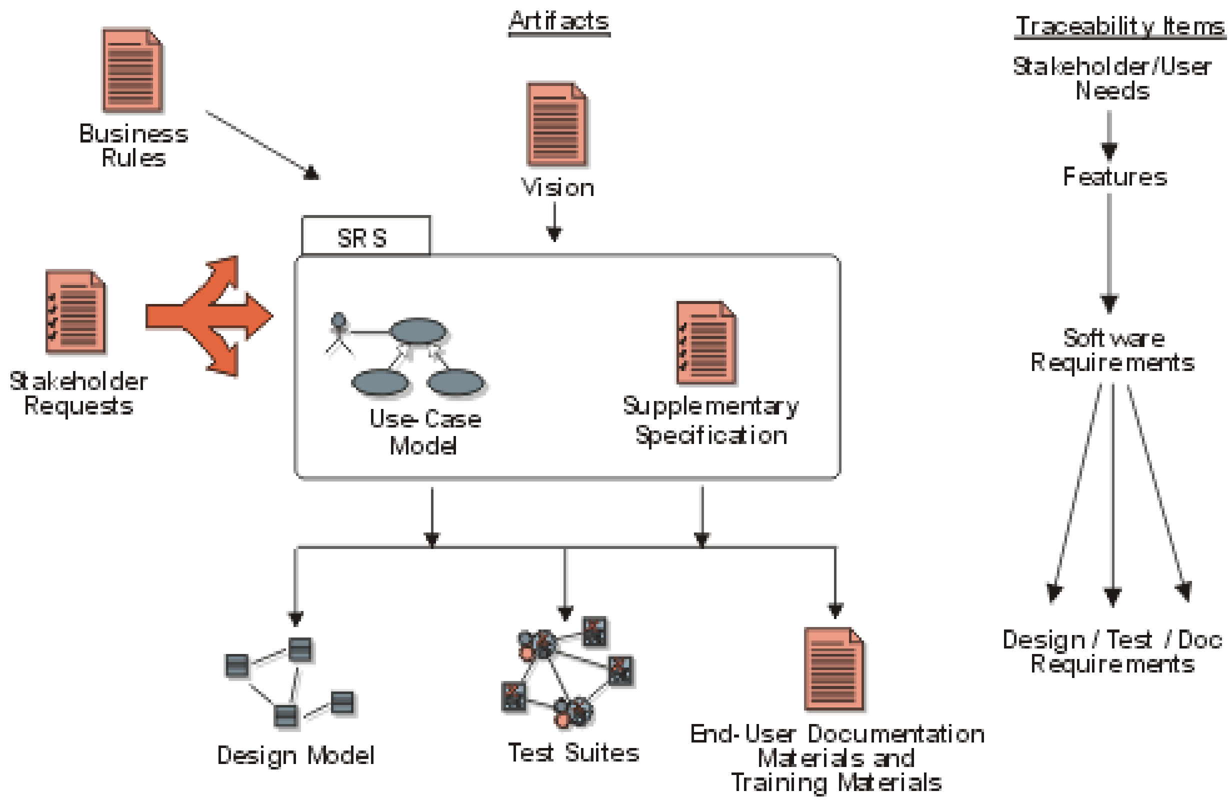
i. Material
ii. Shape
iii. Size
iv. Weight

This type of requirement can be used to represent hardware requirements, such as the physical network configurations required.

3.3 Types of Requirements

Traditionally, requirements are looked upon as statements of text fitting into one of the categories mentioned in **Concepts: Requirements.** Each requirement states "a condition or capability to which the system must conform".

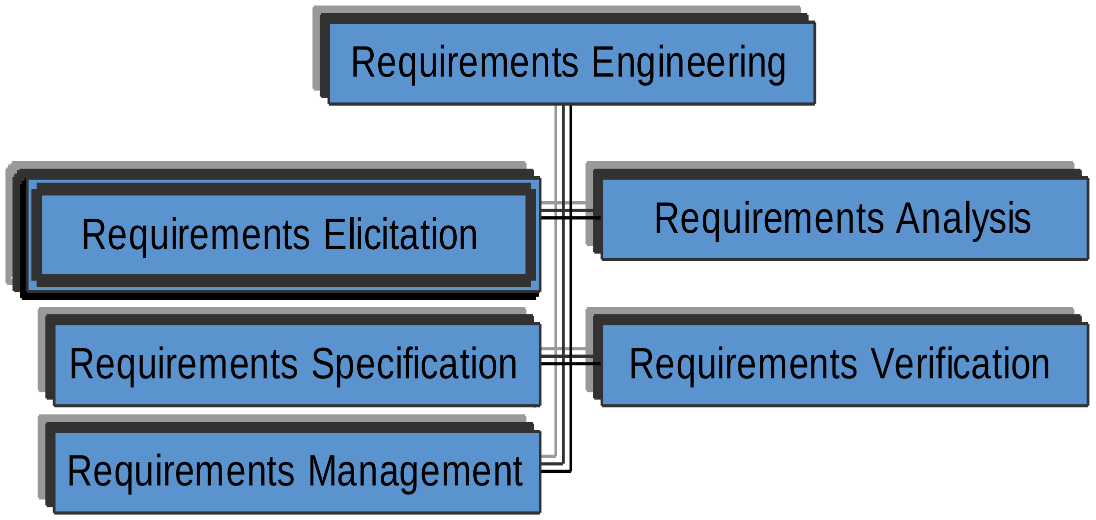
To perform effective **requirements management**, we have learned that it helps to extend what we maintain as requirements beyond only the detailed "software requirements". We introduce the notion of **requirements types** to help separate the different levels of abstraction and purposes of our requirements.



We may want to keep track of ambiguous "wishes", as well as formal requests, from our stakeholders to make sure we know how they are taken care of. The Vision document helps us keep track of key "user needs" and "features" of the system. The use-case model is an effective way of expressing detailed functional "software requirements", therefore use cases may need to be tracked and maintained as requirements, as well as perhaps individual statements within the use case properties which state "conditions or capabilities to which the system must conform". Supplementary Specifications may contain other "software requirements", such as design constraints or legal or regulatory requirements on our system. For a complete definition of the software requirements, use cases and Supplementary Specifications may be packaged together to define a Software Requirements Specification (SRS) for a particular "feature" or other subsystem grouping.

The larger and more intricate the system developed, the more expressions, or types of requirements appear and the greater the volume of requirements. "Business rules" and "vision" statements for a project trace to "user needs", "features" or other "product requirements". Use cases or other forms of modeling and other Supplementary Specifications drive design requirements, which may be further decomposed to functional and non- functional "software requirements" represented in analysis & design models and diagrams.

3.3.1 Requirements Engineering Process3.3.2 Process stages:



It is impossible to define a complete and consistent set of requirements because:

i. New systems improve upon old systems. Also it is impossible to predict effect of a new system upon an organization.

ii. Large systems serve a diverse community. Each person has different requirements and priorities hence the system requirements are a compromise.

iii. End-users often don't write the system requirements. Business decisions influence the design, to the detriment of end-user needs.

3.3.2 Process stages:

1. Feasibility study: This is the quick and dirty evaluation to determine if current technology can solve the problem cost effectively.
2. Requirements analysis: Derive system requirements through study of existing systems, user interviews, task, analysis, etc.
3. Requirements definition: Write-up the analysis results.
4. Requirements specification: Create a detailed and precise set of system requirements. Can be done in parallel with high-level system design. Correct errors.

3.4 Software Requirements Document

Should satisfy the following:

1. Specify external system behaviour.
2. Specify implementation constraints.
3. Easy to change.
4. Serve as reference for system maintainers.
5. Record forethought about the system life cycle.
6. Characterize acceptable responses to undesired events.

A generic structure for the document:

1. Introduction. Why and, briefly, what.
2. Make no assumptions.
3. System models. Show relationships between system, components, environment.
4. Functional requirements definition. Services provided.
5. Non-Functional requirements definition. Constraints and restrictions.
6. System evolution. Fundamental assumptions and anticipated changes.
7. Requirements specification.
8. Hardware.
9. Database. Logical organization of data.

3.5 Requirements Validation

Errors are expensive hence should be checked:

1. Validity. What functionality do the users need?
2. Consistency. Requirements should not contradict.
3. Completeness. All user needed functions should be included.
4. Realism. Don't expect miraculous technological advances.

Some validation mechanisms:

1. Expressing requirements in a formal notation.
2. Prototyping.
3. Requirements reviews.

3.6 Requirements Evolution Enduring requirements.

1. Volatile requirements.

Activity A/ Self Assessment Exercise

i. describe the FURPS+ mode of Requirement engineering categorization
ii. with the aid of a diagram describe requirement engineering process
iii. itemize and explain the ideal content of a software requirement engineering document

3.7 Factors in Requirements engineering

i. Stakeholder identification

A major incidence in requirement engineering is identification of *stakeholders*. It is increasingly recognized that stakeholders are not limited to the organization employing the analyst. Other stakeholders will include those organizations that integrate (or should integrate) horizontally with the organization the analyst is designing the system for, any back office systems or organizations and the Senior management.

ii. Stakeholder interviews

Stakeholder interviews are a common method used in requirement analysis. These interviews may reveal requirements not previously envisaged as being within the scope of the project, and requirements may be contradictory. However, each stakeholder will have an idea of their expectation or will have visualized their requirements.

iii. Contract-style requirement lists

One traditional way of documenting requirements has been contract style requirement lists. In a complex system such requirements lists can run to hundreds of pages.

iv. Measurable goals

Best practices take the composed list of requirements merely as clues and repeatedly ask "why?" until the actual business purposes are discovered. Stakeholders and developers can then devise tests to measure what level of each goal has been achieved thus far. Such goals change more slowly than the long list of specific but unmeasured requirements. Once a small set of critical, measured goals has been established, rapid prototyping and short iterative development phases may proceed to deliver actual stakeholder value long before the project is half over.

v. Prototypes

Prototyping was seen as the solution to the requirements analysis problem. Prototypes are mock-ups of an application. Mock-ups allow users to visualize an application that hasn't yet been constructed. Prototypes help users get an idea of what the system will look like, and make it easier for users to make design decisions without waiting for the system to be built. 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.

However, over the next decade, while proving a useful technique, prototyping did not solve the requirements problem:

a. Managers, once they see a prototype, may have a hard time understanding that the finished design will not be produced for some time.
b. Designers often feel compelled to use patched together prototype code in the real system, because they are afraid to 'waste time' starting again.
c. Prototypes principally help with design decisions and user interface design. However, they cannot tell you what the requirements originally were.
d. Designers and end users can focus too much on user interface design and too little on producing a system that serves the business process.
e. 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 using synthesized functionality. Wireframes are made in a variety of graphic design documents, and often remove all color from the software design (i.e. use a greyscale 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.

vi. Use cases

A use case is a technique for documenting the potential requirements of a new system or software change. Each use case provides one or more *scenarios* that convey how the system should interact with the end user or another system to achieve a specific business goal. Use cases typically avoid technical jargon, preferring instead the language of the end user or *domain expert*. Use cases are often co-authored by requirements engineers and stakeholders.

Use cases are deceptively simple tools for describing the behavior of software or systems. A use case contains a textual description of all of the ways which the intended users could work with the software or system. Use cases do not describe any internal workings of the system, nor do they explain how that system will be implemented. They simply show the steps that a user follows to perform a task. All the ways that users interact with a system can be described in this manner.

3.8 Software requirements specification

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).

Recommended approaches for the specification of software requirements are described by IEEE 830-1998. This standard describes possible structures, desirable contents, and qualities of a software requirements specification.

3.8.1 Types of Requirements

Requirements are categorized in several ways. The following are common categorizations of requirements that relate to technical management:[1]

a .Customer Requirements

Statements of fact and assumptions that define the expectations of the system in terms of mission objectives, environment, constraints, and measures of effectiveness and suitability (MOE/MOS). The customers are those that perform the eight primary functions of systems engineering, with special emphasis on the operator as the key customer. Operational requirements will define the basic need and, at a minimum, answer the questions posed in the following listing:

i. *Operational distribution or deployment*: Where will the system be used?
ii. *Mission profile or scenario*: How will the system accomplish its mission objective?
iii. *Performance and related parameters*: What are the critical system parameters to accomplish the mission?
iv. *Utilization environments*: How are the various system components to be used?
v. *Effectiveness requirements*: How effective or efficient must the system be in performing its mission?
vi. *Operational life cycle*: How long will the system be in use by the user?
vii. *Environment*: What environments will the system be expected to operate in an effective manner?

b. Functional Requirements

Functional requirements explain what has to be done by identifying the necessary task, action or activity that must be accomplished. Functional requirements analysis will be used as the toplevel functions for functional analysis.

c. Non-functional Requirements

Non-functional requirements are requirements that specify criteria that can be used to judge the operation of a system, rather than specific behaviors.

d. Performance Requirements

The extent to which a mission or function must be executed; generally measured in terms of quantity, quality, coverage, timeliness or readiness. During requirements analysis, performance (how well does it have to be done) requirements will be interactively developed across all identified functions based on system life cycle factors; and characterized in terms of the degree of certainty in their estimate, the degree of criticality to system success, and their relationship to other requirements.[1]

e. Design Requirements

The “build to,” “code to,” and “buy to” requirements for products and “how to execute” requirements for processes expressed in technical data packages and technical manuals.

3.8.2 Requirements analysis issues

a. Stakeholder issues

Here are some of the ways users can inhibit requirements gathering:

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

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

b. Engineer/developer issues

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

i. 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.
ii. 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.
iii. Analysis may often be carried out by engineers or programmers, rather than personnel with the people skills and the domain knowledge to understand a client's needs properly.

c. Attempted solutions

One attempted solution to communications problems has been to employ specialists in business or system analysis.

Techniques introduced in the 1990s like prototyping, Unified Modeling Language (UML), use cases, and Agile software development are also intended as solutions to problems encountered with previous methods.

Also, a new class of application simulation or application definition tools have entered the market. These tools are designed to bridge the communication gap between business users and the IT organization — and also to allow applications to be 'test marketed' before any code is produced. The best of these tools offer:

i. Electronic whiteboards to sketch application flows and test alternatives
ii. Ability to capture business logic and data needs
iii. Ability to generate high fidelity prototypes that closely imitate the final application
iv. Interactivity
v. Capability to add contextual requirements and other comments
vi. Ability for remote and distributed users to run and interact with the simulation

3.8.4 Qualities of a Software Requirements Specification

There are many good definitions of System and Software Requirements Specifications that will provide us a good basis upon which we can both define a great specification and help us identify deficiencies in our past efforts. The problem is not lack of knowledge about how to create a correctly formatted specification or even what should go into the specification. The problem is that we don't follow the definitions out there.

We have to keep in mind that the goal is not to create great specifications but to create great products and great software. The IEEE is an excellent source for definitions of System and Software Specifications. For example designers of real-time, embedded system software, use IEEE STD 830-1998 as the basis for all of our Software Specifications unless specifically requested by clients. Essential to having a great Software Specification is having a great System Specification. The equivalent IEEE standard for that is IEEE STD 1233-1998. However, for most purposes in smaller systems, the same templates can be used for both.

3.8.5 Benefits of a Good SRS

The IEEE 830 standard defines the benefits of a good SRS:

a. *Establish the basis for agreement between the customers and the suppliers on what the software product is to do.* The complete description of the functions to be performed by the software specified in the SRS will assist the potential users to determine if the software specified meets their needs or how the software must be modified to meet their needs. This is used as the basis of our contract with our clients all the time.

b. *Reduce the development effort.* The preparation of the SRS forces the various concerned groups in the customer’s organization to consider rigorously all of the requirements before design begins and reduces later redesign, recoding, and retesting. Careful review of the requirements in the SRS can reveal omissions, misunderstandings, and inconsistencies early in the development cycle when these problems are easier to correct.

c. *Provide a basis for estimating costs and schedules.* The description of the product to be developed as given in the SRS is a realistic basis for estimating project costs and can be used to obtain approval for bids or price estimates.

d. *Provide a baseline for validation and verification.* Organizations can develop their validation and Verification plans much more productively from a good SRS. As a part of the development contract, the SRS provides a baseline against which compliance can be measured. This is used to create the Test Plan.

e. *Facilitate transfer.*The SRS makes it easier to transfer the software product to new users or new machines. Customers thus find it easier to transfer the software to other parts of their organization, and suppliers find it easier to transfer it to new customers.

f. *Serve as a basis for enhancement.* Because the SRS discusses the product but not the project that developed it, the SRS serves as a basis for later enhancement of the finished product. The SRS may need to be altered, but it does provide a foundation for continued production evaluation. This is often a major pitfall – when the SRS is not continually updated with changes.

3.8.6 What Should the SRS Address?

Again from the IEEE standard: The basic issues that the SRS writer(s) shall address are the following:

a) *Functionality.* What is the software supposed to do?

b) *External interfaces.* How does the software interact with people, the system's hardware, other hardware, and other software?

c) *Performance.* What is the speed, availability, response time, recovery time of various software functions, etc.?

d) *Attributes.* What are the portability, correctness, maintainability, security, etc. considerations?

e) *Design constraints imposed on an implementation.* Are there any required standards in effect, implementation language, policies for database integrity, resource limits, operating environment(s) etc.?

3.8.7 The Characteristics of a Good SRS

Again based on IEEE standard: An SRS should be

a) Correct b) Unambiguous c) Complete d) Consistent e) Ranked for importance and/or stability

f) Verifiable g) Modifiable h) Traceable

i. Correct - This is the major factor. Of course you want the specification to be correct. No one writes a specification that they know is incorrect. We like to say - "Correct and Ever Correcting." The discipline is keeping the specification up to date when you find things that are not correct.

ii. Unambiguous - An SRS is unambiguous if, and only if, every requirement stated therein has only one interpretation. This is easier said than done. Spending time on this area prior to releasing the SRS can be a waste of time. But as you find ambiguities - fix them.

iii. **Complete -** A simple judge of this is that it should be all that is needed by the software designers to create the software.

iv. **Consistent -** The SRS should be consistent within itself and consistent to its reference documents. If you call an input "Start and Stop" in one place, don't call it "Start/Stop" in another.
v. **Ranked for Importance -** Very often a new system has requirements that are really marketing wish lists. Some may not be achievable. It is useful to provide this information in the SRS.
vi. **Verifiable – avoid ambiguous claims and** requirements like - "It should provide the user a fast response or "The system should never crash." Instead, provide a quantitative requirement like: "Every key stroke should provide a user response within 100 milliseconds."
vii. **Modifiable -** Having the same requirement in more than one place may not be wrong - but tends to make the document not maintainable.
viii. **Traceable -** Often, this is not important in a non-politicized environment. However, in most organizations, it is sometimes useful to connect the requirements in the SRS to a higher level document. Why do we need this requirement?

3.8.8 The Difference Between a System Specification and a Software Specification

Very often we find that companies do not understand the difference between a System specification and a Software Specification. The following is a high level list of requirements that should be addressed in a System Specification:

i. Define the functions of the system
ii. Define the Hardware / Software Functional Partitioning
iii. Define the Performance Specification
iv. Define the Hardware / Software Performance Partitioning
v. Define Safety Requirements
vi. Define the User Interface (A good user’s manual is often an overlooked part of the System specification. Many of our customers haven’t even considered that this is the right time to write the user’s manual.)
vii. Provide Installation Drawings/Instructions.
viii. Provide Interface Control Drawings (ICD’s, External I/O)

One job of the System specification is to define the full functionality of the system. In many systems we work on, some functionality is performed in hardware and some in software. It is the job of the System specification to define the full functionality and like the performance requirements, to set in motion the trade-offs and preliminary design studies to allocate these functions to the different disciplines (mechanical, electrical, software).

Another function of the System specification is to specify performance. For example, if the System is required to move a mechanism to a particular position accurate to a repeatability of ± 1 millimeter, that is a System’s requirement. Some portion of that repeatability specification will belong to the mechanical hardware, some to the servo amplifier and electronics and some to the software. It is the job of the System specification to provide that requirement and to set in motion the partitioning between mechanical hardware, electronics, and software. Very often the System specification will leave this partitioning until later when you learn more about the system and certain factors are traded off (For example, if we do this in software we would need to run the processor clock at 40 mHz. However, if we did this function in hardware, we could run the processor clock at 12 mHz). This implies that a certain level of research or even prototyping and benchmarking needs to be done to create a System spec. It is useful to say that explicitly.

However, for all practical purposes, in most of the small to medium size companies, they combine the software and the systems documents. This is done primarily because most of the complexity is in the software. When the hardware is used to meet a functional requirement, it often is something that the software wants to be well documented. Very often, the software is called upon to meet the system requirement with the hardware you have. Very often, there is not a systems department to drive the project and the software engineers become the systems engineers. For small projects, this is workable even if not ideal. In this case, the specification should make clear which requirements are software, which are hardware, and which are mechanical.

Activity B/Self Assessment Exercise

i. Enumerate the various necessary factors in requirement engineering
ii. Describe the different categories of requirements
iii. What are the attributes of a good SRS

3.8.9 Guidelines to Software Documentation

We have found that taking the time up front pays dividends down stream. If you don’t have time to specify it up front, you probably don’t have the time to do the project.

Here are some of our guidelines:

i. Spend time specifying and documenting well software that you plan to keep.
ii. Keep documentation to a minimum when the software will only be used for a short time or has a limited number of users.
iii. Have separate individuals write the specifications (not the individual who will write the code).
iv. The person to write the specification should have good communication skills.
v. Pretty diagrams can help but often tables and charts are easier to maintain and can communicate the same requirements.
vi. Take your time with complicated requirements. Vagueness in those areas will come back to bite you later.
vii. Conversely, watch out for over-documenting those functions that are well understood by many people but for which you can create some great requirements.
viii. Keep the SRS up to date as you make changes.
ix. Approximately 20-25% of the project time should be allocated to requirements definition.
x. Keep 5% of the project time for updating the requirements after the design has begun.
xi. Test the requirements document by using it as the basis for writing the test plan.

Tutor Marked Assignment

As a consultant system analyst, a small scale industry engaged you to develop an SRS for the new accounting system they are intending to acquire.

- **i.** **Produce an SRS for the accounting software**
- **ii.** **List some documentation guideline that you followed**

4.0 Summary and Conclusion

Many clients do not initially developers with sufficient information upon which to make a reasonable estimate of how big a project is and of how much it is going to cost. In many cases they may not appreciate what information is required. Hence requirement engineering tries to bring all the stakeholders together so avoid developing a product that would be that would not solve the target problem after time and money has been invested. The basic document that is needed is called a Requirements Specification. In other words a description of what you want the system to do. This document may also be called a Business Needs Specification. The Requirements Specification should generally NOT be written in computer terms or contain assumptions about how the system should be written (unless these are part of the requirements).

5.0 References and Further Readings

1. **www.ieee.com**
2. **www.wilkipaedia.com/requirment-engineering/notes**
3. **www.aldexsoftware.com/news**
4. **www.sei.com/lectures/re.html**
5. **www.ibm.com/designs/projects/business-doc.htm**
6.

Table of Contents

1.0 Introduction

People often talk about *processes*; a sequence of *events* that have occurred. Typical descriptive sentences will be of the form "First **this** happens, then **this** happens. After that, if this condition is true, **that** happens". For communicative purposes between people, such sentences suffice, but for computers, a more formal approach is needed. One such formalism is the notion of a *transition network*.

2.0 Objectives

At the end of this unit you are expected to understand:

i. what transition networks and states are
ii. different types of states
iii. how to encode transition networks in XML or any other language
iv. illustrate the entire process using simple diagrammatically

3.0 What are transition networks?

Loosely speaking, a transition network is a set of states and the transitions between them. As noted above, they are good at capturing the notion of process. For example:

i. Control processes such as those in a digitally controlled heating system.
ii. Processes controlling manufacture or design.
iii. Workflow processes such as those found in product data management software.

They are also useful in modeling the behaviour of systems and can be used in object-oriented analysis to create formal models of object interaction and larger system behaviour. Transition networks are closely related to *finite state machines*(FSM) , and to *data flow diagrams*(DFD), but they are augmented with the following capabilities:

i. Transition networks are not limited to "accepting or rejecting their input". Transition networks may execute *actions* or fire off *events* during transitions.
ii. Transition networks can interact with other objects, thereby affecting change in the transition network (or in other networks).
iii. Transitions in transition networks can be controlled by guard conditions that prohibit or allow the transition to be followed.
iv. These guard conditions can be dependent on any predicate involving objects from within the environment of the transition network.

As such, transition networks can be used to describe far more complex interactions or processes than either FSMs or DFDs allow.

3.1 What are states?

A state within a transition network can loosely be defined as a point between transitions. In transition network diagrams, a state is typically depicted as a circle containing the name of the state, as show below.

Depiction of the "data ready" state

While states are a point between transitions, there are some advantages to dealing with them explicitly:

i. Talking about states forms part of normal conversion. For example, people say, "my engine is off" meaning that the engine is in the off state. The principals of natural design call for states to be modeled explicitly.
ii. There can often be multiple transitions into, and more importantly, out of, a state. As such, states can represent a *branch* in a process.
iii. In some transition network formalisms, entry and exit from states can also execute actions and generate events.

Typically states are defined in terms of the attributes of an object, or objects in the system. For example, one would say that water has a number of states:

i. Frozen - when the temperature of the water is less than 0 degrees.
ii. Cold - when the temperature is less than 32 degrees.
iii. Hot - when the temperature is greater than 80 degrees.
iv. Boiled - when the temperature is greater than 100 degrees.
v. Normal - The state is not Frozen, Cold, Hot or Boiled.

The important point about the last example is that it shows one of the requirements of states: that they be exclusive. An object, or system, cannot be in two states at once.

3.1.1 Active and passive states

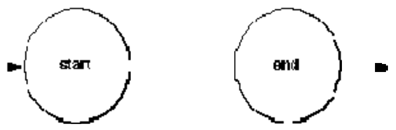
In many processes, there is the notion of continuum, which is not completely captured by the above definition of state. For example, we have the notion of "thawing", where the word implies an ongoing, and *active* state that is continually altering the system. This will be discussed further in the section on transitions.

3.1.2 Start and End states

Even though the process may allow entry and exit from multiple states, when describing the execution of a process, there is always a beginning, and an end. Transition networks, likewise, require start and ending states. Start and end states can be defined as follows:

i. Start state - a state that, in the context of a given execution, has no transitions preceding entry into the state.
ii. End state - a state that has no transitions out of the state (no *forward* transitions).

For the rest of this unit, the following two symbols will be used.

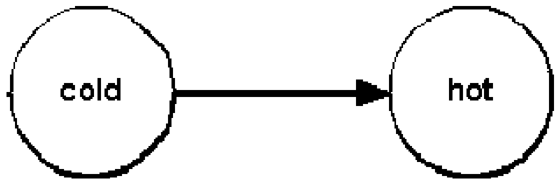


Start and End state diagrams.

Note that this differs from traditional data flow diagram symbols.

3.1.3 What are transitions?

A transition is an atomic, directed connection between one state and another. Typically transitions are represented as a line connecting two states. The following is a simple example.



Sample transition from "cold" to "hot"

Bi-directional transitions (transitions both from a to b and b to a) are typically modeled as two separate transitions.

Transitions can be defined implicitly by the exclusionary nature of states: when the conditions defining one state become false, and the conditions for another true, there is an implicit transition from one state to another; this model has only limited applicability however.

3.1.4 Transitions and Active States

An active state can be looked upon as a state that is being continually updated, or a **gradual transition** from one state to another. As such, active states can be looked upon as a form of continually evaluated transition. While using transitions to model active states does not capture the full semantics (because they are atomic, and not continually updated), for most purposes this model suffices to capture the distinction. Note that in fact it is often useful to think at an even finer level of granularity, and break transitions down into 3 discrete steps that form an atomic action. Those steps are:

i. Leaving a state - the point at which the state has been left, but traversal of the transition between states has not yet begun.
ii. Transitioning between states - the act of traversing the transition between states.
iii. Entering a state - the point at which the new state is entered, but before the "resting point" has been reached.

These 3 steps can, of course, be decomposed into a transition network.

3.2 Conditions, Actions and Events

The main thing separating transition networks from FSM is their active nature. This is captured in the notion of conditions, actions, and events.

3.2.1 Conditions

In order to control the set of transitions that are possible within a transition network, it is necessary to introduce the notion of a *condition* . Essentially a condition is a set of predicates that *guard* entry to and/or exit from a given state. In other words, conditions filter the set of possible transitions. In some systems, the conditions are placed solely on the transitions themselves, in others conditions can be placed upon states as a precondition to entry. These are essentially the same thing, and are not mutually exclusive: in a system allowing both, the union of the guard conditions on the transition and the preconditions on the state form the total set of conditions to be met (i.e. this is an implicit "and" combination of the predicates). Likewise, it is possible to have postconditions on a state controlling the ability to transition from a state.

Given the above, there are three types of conditions found in transition network systems:

i. State preconditions - conditions that guard entry into a state.
ii. State postconditions - conditions that guard exit from a state.
iii. Transition preconditions - conditions guarding the transition from one state to another.

3.2.2 Actions

In transition networks *actions* , or *operations* , allow processing to be associated with a transition. Typically such actions will involve simple interaction with objects in the environment of the transition (getting and setting properties typically), though any arbitrary processing can be performed within the context of the environment. One rule must be obeyed however: altering the environment such that the guard conditions would now fail, should not result in the transition failing. Transitions must always be regarded as atomic.

Typically, actions are associated with the transition, but it is often desirable to be able to specify actions at each of the three steps involved in making a transition. As such, there are three sets of actions that can be invoked:

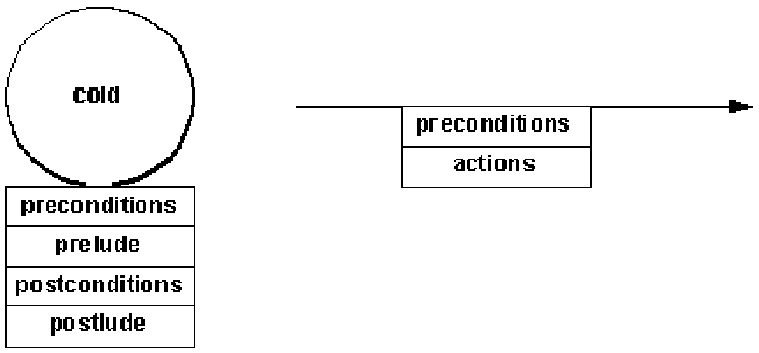
i. State postlude - the set of actions associated with the exit from a state.
ii. Transition body - the set of actions associated with transition traversal.
iii. State prelude - the set of actions associated with the entry into a state.

3.2.3 Events

Actions are scoped to the environment in which they are executed, and so they can only affect local state. *Events* , or *messages*, represent the interaction of the transition network with objects outside the local scope. Typically, they are modeled as actions that have the ability to send an event.

3.2.4 Diagramming Conditions, Actions and Events

In order to capture conditions and actions in a diagram (again, events are modeled as actions capable of sending an event), we must augment the symbols given above. The augmented version shown below will be used throughout the rest of this unit.



Augmented symbols for representing conditions and actions

3.3 Encoding Transition Networks in XML

Transition networks form a graph, and XML is hierarchical, so there would appear to be some discord between the two models. This is not the case if a transition network is seen as a set of states, and a set of transitions: these sets can be easily marked up in XML. Following the practice of natural design leads to a fairly natural mapping from this view of a transition network to its XML representation. This section describes that mapping.

3.3.1 Marking up States

From the explanations in the introduction, states can be modeled as objects with the following type:

```
state = object {
  name           = string;
  preconditions  = set of predicates;
  prelude        = ordered set of actions;
  postconditions = set of predicates;
  postlude       = ordered set of actions;
};
```

Translating this into XML yields a simple element definition as shown below.

```
<!ELEMENT state  (properties?,
                  preconditions?,
                  prelude?,
                  postlude?,
                  postconditions?) >

<!ATTLIST state id   ID    #REQUIRED
                name CDATA #REQUIRED >
```

The `properties`, `preconditions`, `postconditions`, `prelude` and `postlude` elements are defined elsewhere. The two main things to note about this definition of a state are:

i. The `id` attribute - this is declared as an `ID` attribute to guarantee that the state identifier is unique throughout the transition network definition.
ii. The `name` attribute - provides a descriptive name for the state.

Activity A/ Self Assessment Exercise

1. **Explain the following terms:**
   i. **Transition networks**
   ii. **States**
   iii. **Transitions**
2. **Describe the following Conditions, Actions and Events**

3.3.2 Marking up Transitions

Like states, the introduction provides the basis for deriving an object type for transitions. A transition can be modeled as:

```
transition = object {
  from           = oid;
  to             = oid;
  preconditions  = set of predicates;
  actions        = ordered set of actions;
};
```

Translating the object into XML once again yields a straightforward element definition, as shown below.

```
<!ELEMENT transition  (properties?,
                       preconditions?,
                       actions?) >

<!ATTLIST transition id        ID            #REQUIRED
                     name      CDATA         #REQUIRED
                     from      IDREF         #REQUIRED
                     to        IDREF         #REQUIRED
                     mode      (auto|manual) #REQUIRED>
```

As for state definitions, the `properties`, `preconditions`, and `actions` elements are defined elsewhere. The main things to note about this element definition are:

i. The `id` attribute - this is declared as an `ID` attribute to guarantee that the transition identifier is unique throughout the transition network definition.
ii. The `name` attribute - provides a descriptive name for the transition.
iii. The `to` and `from` attributes - these form the connection between states. They are defined as `IDREF` attributes so that they are constrained to reference the `id` attribute of a state (though this constraint is cannot be enforced by the DTD alone).
iv. The `mode` attribute - this states whether the transition should be taken automatically if the set of conditions allows it. If this is set to `auto`, the transition will be taken automatically, if it is set to `manual`, an event of some sort will be needed to force the transition.

3.3.3 Marking up Actions

One of the great challenges in using transition networks is modeling actions. Typically, behaviour in systems is defined in a procedural manner rather than declaratively: in other words, actions are defined in terms of **how** they are to be performed, rather than **what** effect the actions will have. This is largely because, as yet, purely declarative systems that can model complex interactions remain a research problem.

As such, the approach taken here to use an expression language defined in XML, but to model behaviour at a coarse level rather than a fine-grained procedural level. As an example, rather than specifying how an event is sent, a `<send-event>` expression would be defined. Within the context of a given application, these coarse-grained actions, if defined well, can be seen as declarative, and a degree of static analysis becomes possible.

The language chosen here is the XEXPR language, an example of which follows.

```
  <set name="AA" value="AA Value"/>
  <print newline="true"><get name="AA"/></print>
```

3.3.4 Preconditions, Postconditions, Prelude, Postlude and Actions

In the element definitions for states and transitions, a number of other elements were defined. There declarations follow:

```
<!ELEMENT preconditions  (%expression;)*>
<!ELEMENT postconditions (%expression;)*>
<!ELEMENT prelude        (%expression;)*>
<!ELEMENT postlude       (%expression;)*>
<!ELEMENT actions        (%expression;)*>
```

They all use a list of expressions. For conditions, expressions are restricted to predicates. The use of a parameter entity makes it easy to extend the set of possible expressions as the XEXPR language allows elements to be bound to expressions, meaning that the base DTD may not model all elements in the definition.

3.4 Definitions For XML Transition Networks

The following is the complete set of definitions from the DTD for XTND along with explanatory prose.

```
<!DOCTYPE xtnd [

<!ENTITY % def.prop PUBLIC "-//EBT//ELEMENTS Property Set//EN">
%def.prop;

<!ENTITY % def.expr PUBLIC "-//EBT//ELEMENTS XML Expression Language//EN">
%def.expr;
```

This starts the DTD, and pulls in two DTD modules: the property and expression definitions. The public identifier for the DTD is `"-//EBT//DTD Transition Network//EN"`

```
<!ELEMENT preconditions  (%expression;)*>
<!ELEMENT postconditions (%expression;)*>
```

Preconditions and postconditions are simply a set of predicates (expressions that evaluate to a Boolean result). The `expression` parameter entity is defined in the `def.expr` module.

```
<!ELEMENT xtnd (properties?,
               definitions?,
               states,
               transitions?)>
<!ELEMENT definitions (define*)>
```

A transition network can optionally have an arbitrary set of properties associated with it that, depending on the application, could either be used as metadata, or form part of the execution environment of the objects in the system.

In addition, a transition network can optionally have a set of definitions. This is simply a list of `<define>` expressions from the XEXPR language that allow elements to be bound to expressions for use in actions. The heart of the transition network is made up of a required set of states, and an optional set of transitions.

```
<!ELEMENT states (state+)>
<!ATTLIST states start IDREF #REQUIRED>
<!ELEMENT state  (properties?,
                  preconditions?,
                  prelude?,
                  postlude?,
                  postconditions?)>
<!ATTLIST state id   ID    #REQUIRED
                name CDATA #REQUIRED>


<!ELEMENT prelude  (%expression;)*>
<!ELEMENT postlude (%expression;)*>
```

The set of states is simply a list of `<state>` elements. Each state must have a unique `ID` attribute and a descriptive `name` attribute. As part of the state, each state can have an optional set of preconditions, an optional set of postconditions, an optional prelude, and an optional postlude. All of these can contain a list of expressions.

There must be a single start state defined in the network, identified by the `start` attribute on the `<states>` element.

```
<!ELEMENT transitions (transition*) >
<!ELEMENT transition  (properties?,
                       preconditions?,
                       actions?) >
<!ATTLIST transition id         ID            #REQUIRED
                     name       CDATA         #REQUIRED
                     from       IDREF         #REQUIRED
                     to         IDREF         #REQUIRED
                     mode       (auto|manual) #REQUIRED>
<!ELEMENT actions (%expression;)* >
```

The set of transitions is an optional, possibly empty list of `<transition>` elements defining how states can be connected.

Activity B/Self Assessment exercise

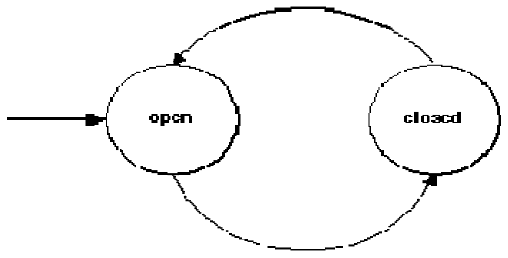
i. **Give a** complete set of definitions from the DTD for XTND along with explanatory prose.
ii. Describe fully how actions could be marked up
iii. What are the three sets of actions to be invoked in making transitions

3.4.1 XTND Examples

The following are a few examples of transition networks marked up using XTND. As can be seen, the modeling is fairly natural and intuitive in practice.

i. Simple transition network

The following shows the transition network for a door, which can have only two states: `open` and `closed`. The door can transition from `open`, to `closed`, and from `closed` to `open`. The door starts off in the `open` state.



Simple transition network for a door

This simple transition network can be straightforwardly marked up in XTND as follows.

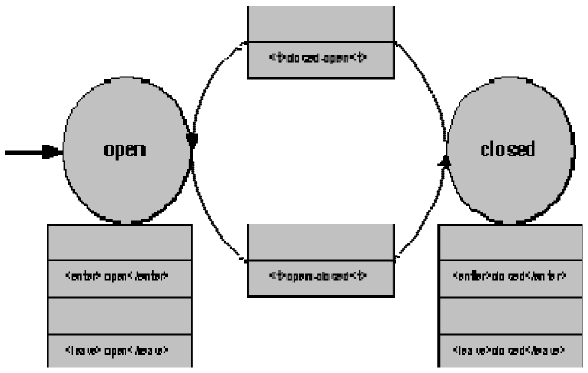
```
<xtnd>
<states start="open">
  <state id="open"   name="Opened State"/>
  <state id="closed" name="Closed State"/>
</states>


<transitions>
  <transition id="open-close"
              name="Transition from opened to closed"
              from="open"
              to="closed"
              mode="manual"/>
  <transition id="close-open"
              name="Transition from closed to opened"
              from="closed"
              to="open"
              mode="manual"/>
</transitions>
</xtnd>
```

Note that the `mode` attribute for both the `open` and `closed` states are set to `manual`. If these were set to `auto`, it would imply the door opening and closing ad infinitum! As it is, some event must cause the transition.

ii. Adding actions

The above can be augmented with actions. In the diagram below we show that an action is associated with the prelude of each state, and with the transitions between states.



Transition network with actions

This can also easily be represented in the XML form, as shown below.

```
<xtnd>
<definitions>
  <define name="enter" args="x">
```

```
    <print>Entering <x/>...</print>
  </define>
  <define name="leave" args="x">
    <print>Leaving <x/>...</print>
  </define>
  <define name="t" args="x">
    <print>Transitioning <x/>...</print>
  </define>
</definitions>

<states start="open">
  <state id="open" name="Opened State">
    <prelude><enter>open</enter></prelude>
    <postlude><leave>open</leave></postlude>
  </state>

  <state id="closed" name="Closed State">
    <prelude><enter>closed</enter></prelude>
    <postlude><leave>closed</leave></postlude>
  </state>
</states>

<transitions>
  <transition id="open-close"
              name="Transition from opened to closed"
              from="open"
              to="closed"
              mode="manual">
    <actions><t>open-closed</t></actions>
  </transition>
  <transition id="close-open"
              name="Transition from closed to opened"
              from="closed"
              to="open"
              mode="manual">
    <actions><t>closed-open</t></actions>
  </transition>
</transitions>
</xtnd>
```

This example also shows the use of definitions, which implies that the expression DTD has also been updated to encompass the defined elements.

3.4.2 Additional DTD Fragments

i. Namespace

The namespace for XTND is http://www.ebt.com/xtnd

People using this namespace are encouraged to use the `xtnd` namespace prefix.

ii.Public Identifier

The public identifier for the XTND DTD is -//EBT//DTD Transition Network/EN

iii. DTD Fragment for Properties

The following is the DTD fragment for properties in the XTND DTD.

```
<!ENTITY % properties "properties">
<!ELEMENT properties (property*)>
<!ELEMENT property (#PCDATA)>
<!ATTLIST property
           name  CDATA                 #REQUIRED
           type  (int|float|boolean|
                  string|object)       #REQUIRED>
```

The constraints placed on the format of the object values are as follows:

int
```
      [+\-]?[0-9]+
```
float
```
      [+\-]? [0-9]+.[0-9]+[+-][eE][0-9]+
```
boolean
```
      true|false
```
string

Any sequence of characters legal as PCDATA. CDATA sections may also be used to escape the textual content.

object

The exact format is application dependent. In Java this might be a BASE64 encoded form of the serialized object.

The public identifier for this fragment is -//EBT//ELEMENTS Property Set//EN

iv. DTD Fragment for Expressions

The following is a DTD fragment for expressions in the XTND DTD. Note that this will need to be extended for each element defined using the `<define>` function.

```
<!ENTITY % expression "bind|define|get|set
                       |expr|add|subtract|multiply
                       |divide|string|integer|float
                       |true|false|nil|eq|neq|leq
                       |geq|lt|gt|and|or|not|if
                       |switch|do|while|print
                       |println">


<!ELEMENT bind EMPTY>
<!ATTLIST bind class CDATA #REQUIRED
               name  CDATA #IMPLIED>

<!ELEMENT define (%expression;)*>
<!ATTLIST define name CDATA #REQUIRED
                 args CDATA #IMPLIED>

<!ELEMENT get EMPTY>
<!ATTLIST get name CDATA #REQUIRED>

<!ELEMENT set (%expression;)*>
<!ATTLIST set name CDATA #REQUIRED>
<!ELEMENT expr (%expression;)*>
<!ELEMENT add (%expression;)*>
```

```
<!ELEMENT subtract (%expression;)*>
<!ELEMENT multiply (%expression;)*>
<!ELEMENT divide (%expression;)*>
<!ELEMENT string (#PCDATA)>
<!ELEMENT integer (#PCDATA)>
<!ELEMENT float (#PCDATA) >
<!ELEMENT true EMPTY>
<!ELEMENT false EMPTY>
<!ELEMENT nil EMPTY>
<!ELEMENT eq (%expression;)*>
<!ELEMENT neq (%expression;)*>
<!ELEMENT leq (%expression;)*>
<!ELEMENT geq (%expression;)*>
<!ELEMENT lt (%expression;)*>
<!ELEMENT gt (%expression;)*>
<!ELEMENT and (%expression;)*>
<!ELEMENT or (%expression;)*>
<!ELEMENT not (%expression;)*>
<!ELEMENT if (%expression;)*>
<!ELEMENT switch   (case*) >
<!ELEMENT case (%expression;)*>
<!ELEMENT do (%expression;)*>
<!ELEMENT while (%expression;)*>
<!ELEMENT print (#PCDATA|%expression;)*>
<!ATTLIST print newline (true|false) #IMPLIED>
<!ELEMENT println (#PCDATA|%expression;)*>
```

The public identifier for this fragment is -//EBT//ELEMENTS XML Expression Language//EN

3.5 State Transition Diagrams

State transition diagram - A diagram consisting of circles to represent states and directed line segments to represent transitions between the states. One or more actions (outputs) may be associated with each transition. The diagram represents a finite state machine. State transition diagrams have been used right from the beginning in object-oriented modeling. The basic idea is to define a machine that has a number of states (hence the term finite state machine). The machine receives events from the outside world, and each event can cause the machine to transition from one state to another. For an example, take a look at figure below. Here the machine is a bottle in a bottling plant. It begins in the empty state. In that state it can receive squirt events. If the squirt event causes the bottle to become full, then it transitions to the full state, otherwise it stays in the empty state (indicated by the transition back to its own state). When in the full state the cap event will cause it to transition to the sealed state. The diagram indicates that a full bottle does not receive squirt events, and that an empty bottle does not receive cap events. Thus you can get a good sense of what events should occur, and what effect they can have on the object. State transition diagrams were around long before object modeling. They give an explicit, even a formal definition of behaviour. A big disadvantage for them is that they mean that you have to define all the possible states of a system. Whilst this is all right for small systems, it soon breaks down in larger systems as there is an exponential growth in the number of states. This state explosion problem leads to state transition diagrams becoming far too complex for much practical use. To combat this state explosion problem, object-oriented methods define separate state-transition diagrams for each class. This pretty much eliminates the explosion problem since each class is simple enough to have a comprehensible state transition diagram. (It does, however, raise a problem in that it is difficult to visualize the behaviour of the whole system from a number of diagrams of individual classes - which leads people to interaction and activity modeling).

The most popular variety of state-transition diagram in object methods is the Harel Statechart as in Figure below. This was introduced by Rumbaugh, taken up by Booch and adopted in the UML. It is one of the more powerful and flexible forms of state transition diagram. A particularly valuable feature of the approach is its ability to generalize states, which allows you to factor out common transitions (thus I can show that the break event applies to both full and empty states by creating the super-state of in-progress). It also has a flexible approach to handling processing. Processes that are instantaneous (i.e. cannot be interrupted) can be bound to the transitions or to the entry or exit of a state, these are called actions. Processes that are long (and can be interrupted) are bound to states, these are called activities. Transitions can also have a condition attached to them, which means that the transition only occurs if the condition is true. There is also a capability for concurrent state diagrams, allowing objects to have more than one diagram to describe their behaviour.

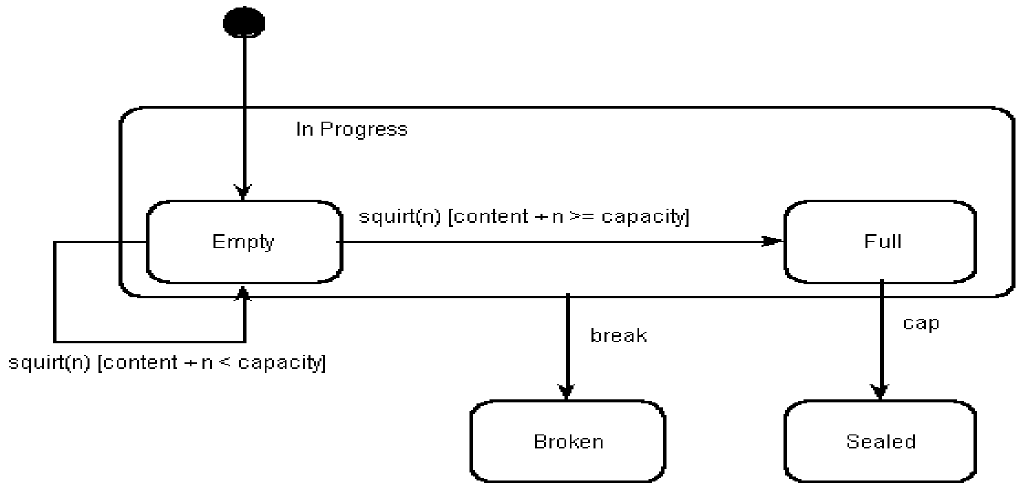

In Progress
Empty
squirt(n) [content + n >= capacity]
Full
break
cap
squirt(n) [content + n < capacity]
Broken
Sealed

Figure 1: A Harel statechart : culled from Harel home page

Not all methods use Harel Statecharts. One of the most notable dissenters is Shlaer/Mellor who use a simpler Moore model state diagram. This form only allows processes to occur when in a state (hence the extra state in Figure 8) and has no superstates. This is a good example of the question of expressiveness in techniques. The Harel Statechart is clearly a more expressive technique, but since it is more expressive there is more to learn when using it. In addition, it is more difficult to implement. Shlaer/Mellor take the view that since state diagrams do not get too complex when drawn for a single object, the extra expressiveness is not worthwhile.

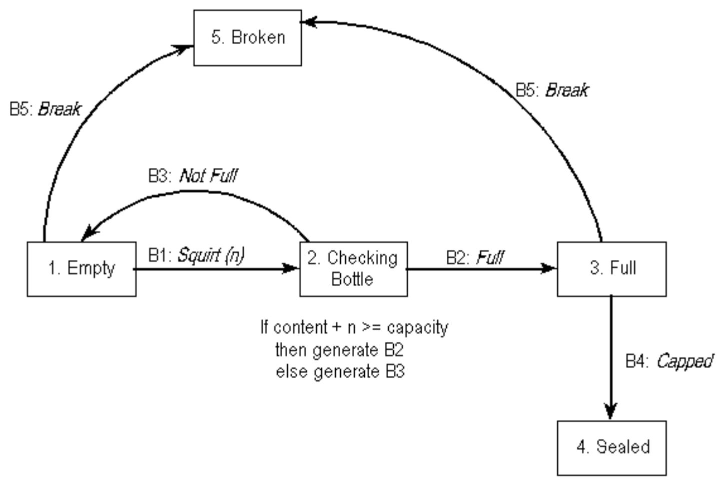


Figure 2: Moore model state diagram as used by Shlaer/Mellor. It is the equivalent to figure above.

3.5.1 When to Use Them

State models are ideal for describing the behaviour of a single object. They are also formal, so tools can be built which can execute them. Their biggest limitation is that they are not good at describing behaviour that involved several objects, for these cases use an interaction diagram or an activity diagram.

People often do not find drawing state diagrams for several objects to be a natural way of describing a process. In these cases you can try either drawing a single state diagram for the process, or using an activity diagram. This defines the basic behaviour, which you then need to refactor to split it across a number of objects.

3.6 Benefits of Using State Charts

State charts are a well known design method to develop embedded systems. We have explained state machines, their visual representation in UML and how to implement them in different programming languages. It is also well known that the later a bug was found in the development process, the more expensive it is to fix.
However, state chart models can be used to increase software quality in the different development phases.

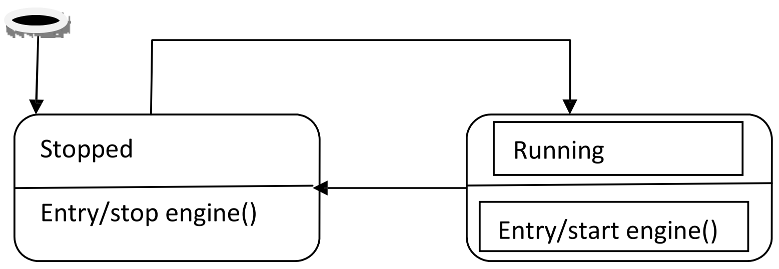


Figure 3: A state chart has a good level of abstraction and is much easier to understand than the corresponding C code.

State charts allow designing the dynamic behaviour of a device: Parts of an embedded device can be often modelled as state machine due to the reactive nature of embedded devices.

Devices react to some kind of external or internal stimuli, which leads to an action and, eventually, to a change of state.

State charts provide a good level of abstraction: Many people with different technical background understand state chart diagrams. This is important for the development of an embedded system, which often involves different engineering disciplines coming together. Therefore, state charts are a very good basis to discuss the modelled behaviour or requirements in design reviews with colleagues or customers. The state chart is much easier to understand.

State charts are useful in all development phases. State charts allow finding defects already in the design phase. To decrease cost of poor quality, it is important to find defects as early as possible in the development process. During the design, defects related to unclear, incomplete or missing requirements are found. Such defects can lead to very costly redesigns or even to the reconstruction of the system if they are found not before the system test. State charts open a number of possibilities to find defects early in the process.

State charts allow simulation of the modelled behaviour. It is easily possible to execute a state chart in a simulator and allow the user to send events to the machine and observe how the state chart reacts to the sent stimuli. This way the user can interactively test the model and improve it where necessary.

Robustness of state charts can be automatically checked on model level. In the hardware design, automatic design rule checks are very common. For software designs, this is not yet common. Software design rules needs to be defined, handmade checks are time-consuming and the result is very dependent on the reviewer. In practice, a tool is needed to ensure that checks are really performed. For UML state charts, the OMG has specified a set of well-formed rules within the UML specification. These rules and number of additional rules can be automatically performed by a model checker.
The following enumeration lists possible rules:

1. State Related:

i. State names must be unique and shall conform to a defined naming schema (e.g. start with a capital S, must not contain spaces to be a valid C-name).

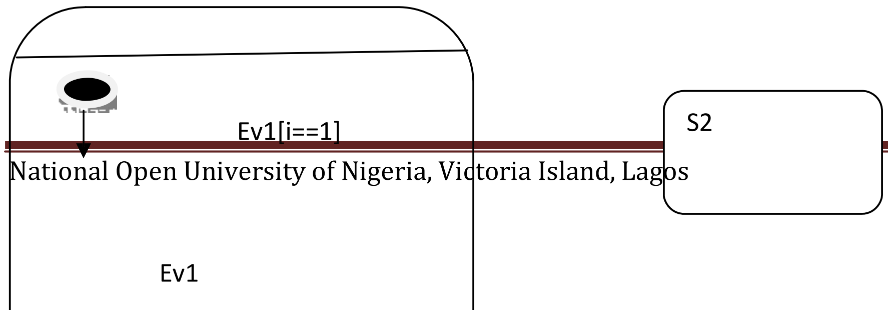
ii. States must be connected by a sequence of transitions outgoing from an initial state (connectivity).

iii. States should have not only incoming or even no transitions at all (isolated states).

iv. Composite states should have more than one child state. If only one child state is defined the composition is superfluous and just creates unnecessary complexity.

v. Initial states must be defined on every state hierarchy and must have exactly one outgoing transition.

vi. Final states must only have incoming transitions.



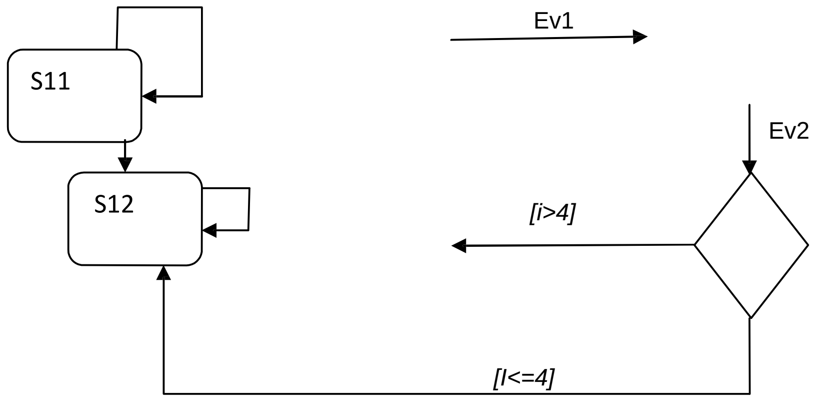


Figure 4: Here's an example of state chart with a number of different model problems.
eetindia.com | EE Times-India

2. Choice Related:

i. A choice must have only one incoming transition.
ii. A choice should have at least two outgoing transitions otherwise it is useless and should be replaced with normal transition.
iii. Every outgoing transition from a choice must have a guard defined.
iv. One default transition must be specified for a choice state, i.e. the guard is defined as 'else' (default from choice).

Some rules seem to be trivial, but are nevertheless very useful to check. Other rules are very difficult to check on source code level (if possible at all) but can be easily checked on model level. **Figure 4** shows a state chart with several defects. A model checker can find them automatically.

They include the following:

i. The state name of the composite state is missing.
ii. There is no default state on the top-level.
iii. S2 is not reachable as transitions triggered by the same event (here ev1) but located deeper in the state hierarchy have priority.
iv. The two transitions triggered by ev11 leaving S11 are ambiguous as one has no guard defined and the other one has.
v. No default transition leaving the choice state is defined.

Automatic code-generation reduces coding errors. Once the state chart was checked, the implementation can start. It is highly recommended not to code the state machine by hand, but let a tool generate the code for you. Automatic code generation has many benefits especially if a model checker is integrated in the generator and can perform a large number of checks automatically. Especially composite state charts can be tricky to code by hand. And latest when transitions or states have to be added because of an additional requirement one wishes to have a generator at hand taking over all the error-prone placement of entry, exit and action code associated with states or transitions. It is important of course that the code generator was developed for the embedded domain. (i.e. it produces efficient but readable code which can be understood from the developer). A number of requirements for such a generator were listed in.
Automatic code generation does not make source code analysis (SCA) needless. SCA technology has been evolving for more than two decades, and well known tools such as PCLint are used by embedded software developers in virtually every industry. Both model

checking and SCA complement each other quite well. It is important to mention that a code generator shall generate code that is not in conflict with used SCA tools.

Test cases can be automatically derived from the model. The beauty of coding even simple algorithms as state machines is that the test plan almost writes itself. All you have to do is to go through every state transition. ***Do this with a highlighter in hand, crossing off the arrows on the state transition diagram as they successfully pass their tests***. This requires a high level of patience , because even a mid-size state machine can have 100 different transitions. However, the number of transitions is an excellent measure of the system's complexity." Based on a state chart model, a tool can take over this time consuming manual task of defining routes through the state chart ensuring 100 per cent transition coverage. Implementing the test cases is a task that you still have to do. But a good code generator can also support you there by automatically generate trace code if you need it for test purposes or post mortem analysis.
For the trivial example in Figure1, above a path with 100 per cent coverage is as follows:
Transition Coverage:
0: From Stopped taking event
start[enoughFuel()] ending in
Running
1: From Running taking event stop ending in Stopped
Transition Coverage for suggested path(s): 100%
There are many good reasons to create state chart models in your next project and use them throughout the development process. Several good general purpose UML tools exist to efficiently model state machines. (It is not recommended to just draw them with a general purpose drawing tool).

The model can then be used for design reviews, advanced model checking, code- and test case generation as described. I strongly vote for automatic source code generation based on the graphical UML model and not creating code directly by hand. This ensures consistency between design and code.

4.0 Conclusion

In many systems, transition networks are used to describe a set of states and the transitions that are possible between them. Common examples are such things as ATM control flows, editorial review processes, and definitions of protocol states. Typically, each of these transition networks has its own specific data format, and its own specific editing tool. Given the rapid transition to pervasive networking, and to application integration and interchange, a standard format for transition networks is desirable. This unit defines such an interchange format, defined in XML: the interchange language for the Internet.

5.0 Summary

i. Transition network is a set of states and the transitions between them; they are good at capturing the notion of process.

ii. They are also useful in modeling the behaviour of systems and can be used in object-oriented analysis to create formal models of object interaction and larger system behaviour.

iii. One of the great challenges in using transition networks is modeling actions. Typically, behaviour in systems is defined in a procedural manner rather than declaratively: in other words, actions are defined in terms of **how** they are to be performed, rather than **what** effect the actions will have.

iv. **State transition diagram -** A diagram consisting of circles to represent states and directed line segments to represent transitions between the states.

v. A big disadvantage for state transition diagrams is that they mean that you have to define all the possible states of a system. Whilst this is all right for small systems, it soon breaks down in larger systems as there is an exponential growth in the number of states

6.0 Tutor Marked Assignment

1. Use Harel State chart to describe the concept of state transition
2. Compare and contrast Moore model state diagram and Harel statechart
3. Briefly explain when to use Moore model state diagram and Harel statechart

7.0Further Readings and Resources

1. J. Martin and C. McClure Diagramming Techniques for Analysts and Programmers

2.Dennis de Chapeaux, Douglas Lea, Penelope Faure Object Oriented System Development

3. http://www.w3.org/XML
4. http://www.w3.org/TR/2000/NOTE-xtnd
5. http://www.w3.org/TR/xtnd
6. http://www. eetindia.com/doc/statechart

1.0 Introduction

Project planning is a discipline for stating how a project will be accomplished within a certain timeframe and budget. Often project planning is ignored in favour of getting on with the work. However, many people fail to realise the value of a project plan in saving time, money and many problems. This unit looks at the steps for creating a simple plan at the beginning of a project. Project planning is part of project management, which relates to the use of schedules such as Gantt charts to plan and subsequently report progress within the project environment. Initially, the project scope is defined and the appropriate methods for completing the project are determined. Following this step, the durations for the various tasks necessary to complete the work are listed and grouped into a work breakdown structure. The logical dependencies between tasks are defined using an activity network diagram that enables identification of the critical path. Float or slack time in the schedule can be calculated using project management software. Then the necessary resources can be estimated and costs for each activity can be allocated to each resource, giving the total project cost. At this stage, the project plan may be optimized to achieve the appropriate balance between resource usage and project duration to comply with the project objectives.

2.0 Objectives

By the end of this unit, you would be able to:

(a) Understand the intricacies of project planning
(b) How to carry out project specifications
(c) Understand the benefits of project specifications and how to locate errors in a project specification document
(d) Understand the art of planning
(e) **Understand the application project life cycle**

3.0 Project Specification

Before describing the role and creation of a specification, we need to introduce and explain a fairly technical term, a numbty. A *numbty* is a person whose brain is totally numb. In this context, numb means "deprived of feeling or the power of unassisted activity"; in general, a numbty needs a high level of stimulation even to get to the right office in the morning. Communication with numbties is severely hampered by the fact that although they think they know what they mean (which they do not), they seldom actually say it, and they never write it down. According to Goldratt Eliyahu,"the main employment of numbties world-wide is in creating project specifications". You must know this as a software project manager- and protect your team accordingly.

A specification is the definition of your project: a statement of the problem, not the solution. Normally, the specification contains errors, ambiguities, misunderstandings and enough rope to hang you and your entire team. Thus before you embark upon the next six months of activity working on the wrong project, you must assume that a numbty was the chief author of the specification you received and you must read, worry, revise and ensure that everyone concerned with the project (from originator, through the workers, to the end-customer) is working with the same understanding. The outcome of this deliberation should be a *written* definition of what is required, by when; and this must be *agreed* by all involved. There are no short-cuts to this; if you fail to spend the time initially, it will cost you far more later on.

3.1.1 Benefits of *written* specification *agreement*

(a) The clarity will reveal misunderstandings
(b) The completeness will remove contradictory assumptions
(c) The rigour of the analysis will expose technical and practical details which numbties normally gloss over through ignorance or fear
(d) The agreement forces all concerned to actually read and think about the details

The work on the specification can be seen as the first stage of Quality Assurance since you are looking for and countering problems in the very foundation of the project - from this perspective the creation of the specification clearly merits a large investment of time.

From a purely defensive point of view, the agreed specification also affords you protection against the numbties who have second thoughts, or new ideas, half way through the project. Once the project is underway, changes cost time (and money). The existence of a demonstrably-agreed specification enables you to resist or to charge for (possibly in terms of extra time) such changes. Further, people tend to forget what they originally thought; you may need proof that you have been working as instructed.

3.1.2 Where to locate errors in a Project Specification

(a) The Global Context: Numbties often focus too narrowly on the work of one team and fail to consider how it fits into the larger picture. Some of the work given to you may actually be undone or duplicated by others. Some of the proposed work may be incompatible with that of others.
(b) The Interfaces: Between your team and both its customers and suppliers, there are interfaces. At these points something gets transferred. Exactly what, how and when should be discussed and agreed from the very beginning. Never assume a common understanding, because you will be wrong. All it takes for your habitual understandings to evaporate is the arrival of one new member, in either of the teams. Define and agree your interfaces and maintain a friendly contact throughout the project.
(c) Time-Scales: Numbties always underestimate the time involved for work. If there are no time-scales in the specification, you can assume that one will be imposed upon you (which will be impossible). You must add realistic dates. The detail should include a precise understanding of the extent of any intermediate stages of the task, particularly those which have to be delivered.
(d) External Dependencies: Your work may depend upon that of others. Make this very clear so that these people too will receive warning of your needs. Highlight the effect that problems with these would have upon your project so that everyone is quite clear about their importance. To be sure, contact these people yourself and ask if they are able to fulfil the assumptions in your specification.
(e) Resources: The numbty tends to ignore resources. The specification should identify the materials, equipment and manpower which are needed for the project. The agreement should include a commitment by your managers to allocate or to fund

them. You should check that the actual numbers are practical and/or correct. If they are omitted, add them - there is bound to be differences in their assumed values.

This seems to make the specification sound like a long document. It should not be. Each of the above could be a simple sub-heading followed by either bullet points or a table - you are not writing a brochure, you are stating the definition of the project in clear, concise and unambiguous terms.

Of course, the specification may change. If circumstances or simply your knowledge change then the specification will be out of date. You should not regard it as cast in stone but rather as a display board where everyone involved can see the current, common understanding of the project. If you change the content everyone must know, but do not hesitate to change it as necessary.

Activity A / Self assessment Exercise

i. what is project planning
ii. describe project specification process
iii. what are the benefits of written specification agreement
iv. explain the five points to locate errors in project specification

3.2 PROVIDING STRUCTURE

Having decided what the specification intends, your next problem is to decide what you and your team actually need to do, and how to do it. As a manager, you have to provide some form of framework both to plan and to communicate what needs doing. Without a structure, the work is a series of unrelated tasks which provides little sense of achievement and no feeling of advancement. If the team has no grasp of how individual tasks fit together towards an understood goal, then the work will seem pointless and they will feel only frustration.

To take the planning forward, therefore, you need to turn the specification into a complete set of tasks with a linking structure. Fortunately, these two requirements are met at the same time since the derivation of such a structure is the simplest method of arriving at a list of tasks.

3.2.1 Work Breakdown Structure

Once you have a clear understanding of the project, and have eliminated the vagaries of the numbties, you then describe it as a set of simpler separate *activities*. If any of these are still too complex for you to easily organise, you break them down also into another level of simpler descriptions, and so on until you can manage everything. Thus your one complex project is organised as a set of simple tasks which together achieve the desired result.

The reasoning behind this is that the human brain (even yours) can only take in and process so much information at one time. To get a real grasp of the project, you have to think about it in pieces rather than trying to process the complexity of its entire details all at once. Thus each level of the project can be understood as the amalgamation of a few simply described smaller units.

In planning any project, you follow the same simple steps: if an item is too complicated to manage, it becomes a list of simpler items. People call this producing a *work breakdown* *structure* to make it sound more formal and impressive. Without following this formal approach you are unlikely to remember all the niggling little details; with this procedure, the details are simply displayed on the final lists.

Note, one common fault is to produce too much detail at the initial planning stage. You should stop when you have a sufficient description of the activity to provide a clear instruction for the person who will actually do the work, and to have a reasonable estimate for the total time/effort involved. You need the former to allocate (or delegate) the task; you need the latter to finish the planning.

3.2.2 Task Allocation

The next stage is a little complicated. You now have to allocate the tasks to different people in the team and, at the same time, order these tasks so that they are performed in a sensible sequence.

Task allocation is not simply a case of handing out the various tasks on your final lists to the people you have available; it is far more subtle (and powerful) than that. As a manager you have to look far beyond the single project; indeed any individual project can be seen as merely a single step in your team's development. The allocation of tasks should thus be seen as a means of increasing the skills and experience of your team - when the project is done, the team should have gained.

In simple terms, consider what each member of your team is capable of and allocate sufficient complexity of tasks to match that (and to slightly stretch). The tasks you allocate are *not* the ones on your finals lists, they are adapted to better suit the needs of your team's development; *tasks are moulded to fit people*, which is far more effective than the other way around. For example, if Eze is to learn something new, the task may be simplified with responsibility given to another to guide and check the work; if Chux is to develop, sufficient tasks are combined so that his responsibility increases beyond what she has held before; if Ebere lacks confidence, the tasks are broken into smaller units which can be completed (and commended) frequently.

Sometimes tasks can be grouped and allocated together. For instance, some tasks which are seemingly independent may benefit from being done together since they use common ideas, information and talents. One person doing them both removes the start-up time for one of them; two people (one on each) will be able to help each other.

The ordering of the tasks is really quite simple, although you may find that sketching a sequence diagram helps you to think it through (and to communicate the result). *Pert charts* are the accepted outcome, but sketches will suffice. Getting the details exactly right, however, can be a long and painful process, and often it can be futile. The degree to which you can predict the future is limited, so too should be the detail of your planning. You must have the broad outlines by which to monitor progress, and sufficient detail to assign each task when it needs to be started, but beyond that - stop and do something useful instead.

3.2.3 Guesstimation

At the initial planning stage the main objective is to get a *realistic* estimate of the time involved in the project. You must establish this not only to assist higher management with their planning, but also to protect your team from being expected to do the impossible. The most important technique for achieving this is known as: *guesstimation*.

Guesstimating schedules is notoriously difficult but it is helped by two approaches:

a) Make your guesstimates of the simple tasks at the bottom of the work break down structure and look for the longest path through the sequence diagram
b) Use the experience from previous projects to improve your guesstimating skills

The corollary to this is that you should keep records in an easily accessible form of all projects as you do them. Part of your final project review should be to update your personal data base of how long various activities take. Managing this planning phase is vital to your success as a manager.

Some people find guesstimating a difficult concept in that if you have no experience of an activity, how can you make a worthwhile estimate? Let us consider such a problem:

How long would it take you to walk all the way from Bar beach Victoria Island to National theatre Iganmu Lagos? Presuming you have never actually tried this (most people take bus part of the way), you really have very little to go on. Indeed if you have actually been to one (and only one) of these places, think about the other. Your job depends upon this, so think carefully. One idea is to start with thinking of the number of kilometers - guess that if you can. Notice, you do not have to be right, merely reasonable. Next, consider the sort of pace you could maintain while trekking for a long time. Now imagine yourself at the beach you do know, and estimate a) how many steps there are to take to get there, and b) how long it takes you to trek there (at that steady pace). To complete, apply a little mathematics.

Now examine how confident you are with this estimate. Assuming you found yourself in Lagos and tried it, you would probably be mildly surprised if you trekked the distance to National theatre in less than half the estimated time and if it took you more than double you would be mildly annoyed. If it took you less than a tenth the time, or ten times as long, you would extremely be surprised/annoyed. In fact, you do not currently believe that that would happen (no really, do you?). The point is that from very little experience of the given problem, you can actually come up with a working estimate - and one which is far better than no estimate at all when it comes to deriving a schedule. Guesstimating does take a little practice, but it is a very useful skill to develop.

3.2.3.1Practical Problems in Guesstimation.

a) You are simply too optimistic. It is human nature at the beginning of a new project to ignore the difficulties and assume best case scenarii - in producing your estimates (and using those of others) you must inject a little realism. In practice, you should also build-in a little slack to allow yourself some tolerance against mistakes. This is known as *defensive scheduling*. Also, if you eventually deliver ahead of the agreed schedule, you will be loved.

b) You will be under pressure from senior management to deliver quickly, especially if the project is being sold competitively. Resist the temptation to rely upon speed as the only selling point. You might, for instance, suggest the criteria of: fewer errors, history of adherence to initial schedules, previous customer satisfaction.

3.3 ESTABLISHING CONTROLS

When the planning phase is over (and agreed), the "doing" phase begins. Once it is in motion, a project acquires a direction and momentum which is totally independent of anything you predicted. If you come to terms with that from the start, you can then enjoy the roller-coaster which follows. To gain some hope, however, you need to establish at the start (within the plan) the means to monitor and to influence the project's progress.

There are two key elements to the control of a project

a) Milestones (clear, unambiguous targets of what, by when)
b) Established means of communication

3.3.1 For you, the milestones are a mechanism to monitor progress; for your team, they are short-term goals which are very tangible. The milestones maintain the momentum and encourage effort; they allow the team to judge their own progress and to celebrate achievement throughout the project rather than just at its end.

The simplest way to construct milestones is to take the timing information from the work breakdown structure and sequence diagram. When you have guesstimated how long each sub-task will take and have strung them together, you can identify by when each of these tasks will actually be completed. This is simple and effective; however, it lacks creativity.

3.3.2 A second method is to construct more significant milestones. These can be found by identify stages in the development of a project which are recognisable as steps towards the final product. Sometimes these are simply the higher levels of your structure; for instance, the completion of a market-evaluation phase. Sometimes, they cut across many parallel activities; for instance, a prototype of the eventual product or a mock-up of the new brochure format.

If you are running parallel activities, this type of milestone is particularly useful since it provides a means of pulling together the people on disparate activities, and so:

a) They all have a shared goal (the common milestone)
b) Their responsibility to (and dependence upon) each other is emphasised
c) Each can provide a new (but informed) viewpoint on the others' work
d) The problems to do with combining the different activities are highlighted and discussed early in the implementation phase
e) You have something tangible which senior management (and numbties) can recognise as progress
f) You have something tangible which your team can celebrate and which constitutes a short-term goal in a possibly long-term project
g) It provides an excellent opportunity for quality checking and for review

Of course, there are milestones and there are mill-stones. You will have to be sensitive to any belief that working for some specific milestone is hindering rather than helping the work forward. If this arises then either you have chosen the wrong milestone, or you have failed to communicate how it fits into the broader structure.

3.3.3 Communication:

Communication is your everything. To monitor progress, to receive early warning of danger, to promote cooperation, to motivate through team involvement, all of these rely upon communication. Regular reports are invaluable - if you clearly define what information is needed and if you teach your team how to provided it in a rapidly accessible form. Often these reports merely say "progressing according to schedule". These you send back, for while the message is desired the evidence is missing: you need to insist that your team monitor their own progress with concrete, tangible, measurements and if this is done, the figures should be included in the report. However, the real value of this practice comes when progress is not according to schedule - then your communication system is worth all the effort you invested in its planning.

Activity B / self assessment Exercise

i. describe the following concepts in software project planning

(a). work breakdown (b) Task allocation (c) Guesstimation

ii. list and explain the problems encountered in guesstimation

iii. describe the two key control elements in software project planning

3.4 THE ARTISTRY IN PLANNING

At the planning stage, you can deal with far more than the mere project at hand. You can also shape the overall pattern of your team's working using the division and type of activities you assign.

3.4.1 Who know best?

Ask your team. They too must be involved in the planning of projects, especially in the lower levels of the work breakdown structure. Not only will they provide information and ideas, but also they will feel ownership in the final plan.

This does not mean that your projects should be planned by committee - rather that you, as manager, plan the project based upon all the available experience and creative ideas. As an initial approach, you could attempt the first level(s) of the work breakdown structure to help you communicate the project to the team and then ask for comments. Then, using these, the final levels could be refined by the people to whom the tasks will be allocated. However, since the specification is so vital, *all* the team should vet the penultimate draft.

3.4.2 Dangers in review

There are two pitfalls to avoid in project reviews:

a) They can be too frequent
b) They can be too drastic

The constant trickle of new information can lead to a vicious cycle of planning and revising which shakes the team's confidence in any particular version of the plan and which destroys the very stability which the structure was designed to provide. You must decide the balance. Pick a point on the horizon and walk confidently towards it. Decide objectively, and explain beforehand, when the review phases will occur and make this a scheduled milestone in itself.

Even though the situation may have changed since the last review, it is important to recognise the work which has been accomplished during the interim. Firstly, you do not want to abandon it since the team will be demotivated feeling that they have achieved nothing. Secondly, this work itself is part of the new situation: it has been done, it should provide a foundation for the next step or at least the basis of a lesson well learnt. Always try to build upon the existing achievements of your team.

3.4.3 Testing and Quality

No plan is complete without explicit provision for testing and quality. As a wise manager, you will know that this should be part of each individual phase of the project. This means that no activity is completed until it has passed the objectively defined criteria which establishes its quality, and these are best defined objectively at the beginning as part of the planning.

When devising the schedule therefore you must include allocated time for this part of each activity. Thus your question is not only: "how long will it take", but also: "how long will the testing take". By asking both questions together you raise the issue of "how do we know we have done it right" at the very beginning and so the testing is more likely to be done in parallel with the implementation. You establish this philosophy for your team by include testing as a justified required cost.

3.4.4 Fitness for purpose

Another reason for stating the testing criteria at the beginning is that you can avoid futile quests for perfection. If you have motivated your team well, they will each take pride in their work and want to do the best job possible. Often this means polishing their work until is shines; often this wastes time. If it is clear at the onset exactly what is needed, then they are more likely to stop when that has been achieved. You need to avoid generalities and to stipulate boundaries; not easy, but essential.

The same is also true when choosing the tools or building-blocks of your project. While it might be nice to have use of the most modern versions, or to develop an exact match to your needs; often there is an old/existing version which will serve almost as well (sufficient for the purpose), and the difference is not worth the time you would need to invest in obtaining or developing the new one. Use what is available whenever possible unless the *difference* in the new version is worth the time, money and the initial, teething pains.

A related idea is that you should discourage too much effort on aspects of the project which are idiosyncratic to that one job. In the specification phase, you might try to eliminate these through negotiation with the customer; in the implementation phase you might leave these parts until last. The reason for this advice is that a *general* piece of work can be tailored to many specific instances; thus, if the work is in a general form, you will be able to rapidly re-use it for other projects. On the other hand, if you produce something which is cut to fit exactly one specific case, you may have to repeat the work entirely even though the next project is fairly similar. At the planning phase, a manager should bare in mind the future and the long-term development of the team as well as the requirements of the current project.

3.4.5 Fighting for time

As a manager, you have to regulate the pressure and work load which is imposed upon your team; you must protect them from the unreasonable demands of the rest of the company. Once you have arrived at what you consider to be a realistic schedule, fight for it. Never let the outside world deflect you from what you know to be practical. If they impose a deadline upon you which is impossible, *clearly* state this and give your reasons. You will need to give some room for compromise, however, since a flat **NO** will be seen as obstructive. Since you want to help the company, you should look for alternative positions.

You could offer a prototype service or product at an earlier date. This might, in some cases, be sufficient for the customer to start the next stage of his/her own project on the understanding that your project would be completed at a later date and the final version would then replace the prototype.

The complexity of the product, or the total number of units, might be reduced. This might, in some cases, be sufficient for the customer's immediate needs. Future enhancements or more units would then be the subject of a subsequent negotiation which, you feel, would be likely to succeed since you will have already demonstrate your ability to deliver on time.

You can show on an alternative schedule that the project could be delivered by the deadline if certain specified resources are given to you or if other projects are rescheduled. Thus, you provide a clear picture of the situation and a possible solution; it is up to your manager then how he proceeds.

3.4.6 Planning for error

The most common error in planning is to assume that there will be no errors in the implementation: in effect, the schedule is derived on the basis of "if nothing goes wrong, this will take ...". Of course, recognising that errors will occur is the reason for implementing a monitoring strategy on the project. Thus when the inevitable does happen, you can react and adapt the plan to compensate. However, by carefully considering errors in advance you can make changes to the original plan to enhance its tolerance. Quite simply, your planning should include time where you stand back from the design and ask: "*what can go wrong?*"; indeed, this is an excellent way of asking your team for their analysis of your plan.

You can try to predict where the errors will occur. By examining the activities' list you can usually pinpoint some activities which are risky (for instance, those involving new equipment) and those which are quite secure (for instance, those your team has done often before). The risky areas might then be given a less stringent time-scale - actually planning-in time for the mistakes. Another possibility is to apply a different strategy, or more resources, to such activities to minimise the disruption. For instance, you could include training or consultancy for new equipment, or you might parallel the work with the foundation of a fall-back position.

3.4.6 Post-mortem

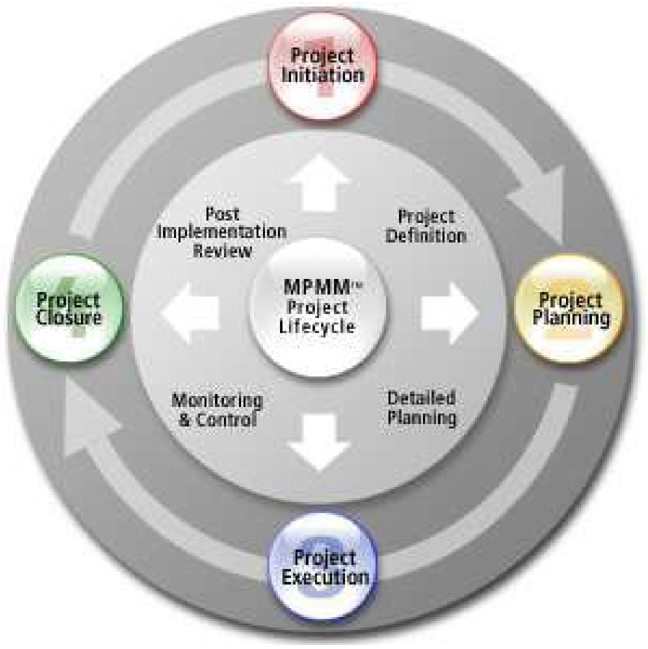
At the end of any project, you should allocate time to reviewing the lessons and information on both the work itself and the management of that work: an open meeting, with open discussion, with the whole team and all customers and suppliers. If you think that this might be thought a waste of time by your own manager, think of the effect it will have on future communications with your customers and suppliers.

3.5 PLANNING FOR THE FUTURE

With all these considerations in merely the "planning" stage of a project, it is perhaps surprising that projects get done at all. In fact projects do get done, but seldom in the predicted manner and often as much by brute force as by careful planning. The point, however, is that this method is non-optimal. Customers feel let down by late delivery, staff are demotivated by constant pressure for impossible goals, corners get cut which harm your reputation, and each project has to overcome the same problems as the last.

With planning, projects can run on time and interact effectively with both customers and suppliers. Everyone involved understands what is wanted and emerging problems are seen and dealt with long before they cause damage. If you want your projects to run this way - then you must invest time in planning.

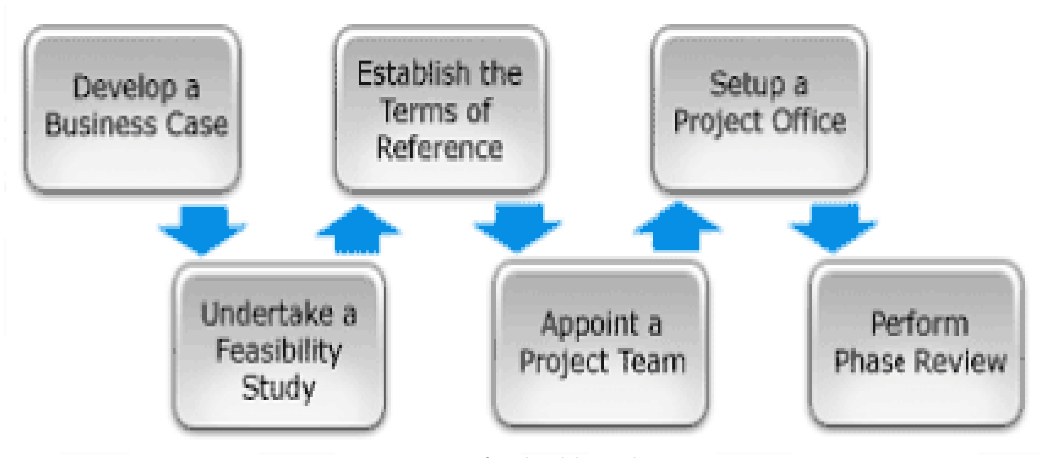
3.5.1 Project Management Life CycleThe Project Life Cycle refers to a logical sequence of activities to accomplish the project's goals or objectives. Regardless of scope or complexity, any project goes through a series of stages during its life. There is first an Initiation or Birth phase, in which the outputs and critical success factors are defined, followed by a Planning phase, characterized by breaking down the project into smaller parts/tasks, an Execution phase, in which the project plan is executed, and lastly a Closure or Exit phase, that marks the completion of the project. Project activities must be grouped into phases because by doing so, the project manager and the core team can efficiently plan and organize resources for each activity, and also objectively measure achievement of goals and justify their decisions to move ahead, correct, or terminate. It is of great importance to organize project phases into industry-specific project cycles. Why? Not only because each industry sector involves specific requirements, tasks, and procedures when it comes to projects, but also because different industry sectors have different needs for life cycle management methodology. And paying close attention to such details is the difference between doing things well and excelling as project managers. Diverse project management tools and methodologies prevail in the different project cycle phases. Let's take a closer look at what's important in each one of these stages:



Adapted from Michigan GIS Project system

3.5.1.1 Project Initiation

Project Initiation is the first phase in the Project Life Cycle and essentially involves starting up the project. You initiate a project by defining its purpose and scope, the justification for initiating it and the solution to be implemented. In this first stage, the scope of the project is defined along with the approach to be taken to deliver the desired outputs. The project manager is appointed and in turn, he selects the team members based on their skills and experience. The most common tools or methodologies used in the initiation stage are Project Charter, Business Plan, Project Framework (or Overview), Business Case Justification, and Milestones Reviews. You will also need to recruit a suitably skilled project team, set up a Project Office and perform an end of Phase Review. The Project Initiation phase involves the following six key steps:

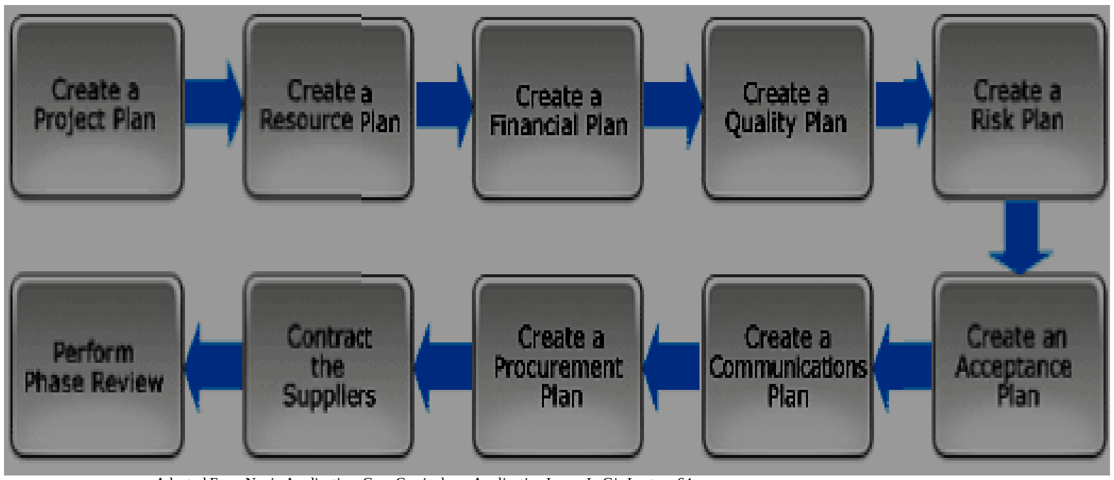


Michigan State GIS Project System

It also involves starting up the project, by documenting a business case, feasibility study, terms of reference, appointing the team and setting up a Project Office.

3.5.1.2 Project Planning

The second phase should include a detailed identification and assignment of each task until the end of the project. It should also include a risk analysis and a definition of a criteria for the successful completion of each deliverable. The governance process is defined, stake holders identified and reporting frequency and channels agreed. The most common tools or methodologies used in the planning stage are Business Plan and Milestones Reviews. After defining the project and appointing the project team, you're ready to enter the detailed Project Planning phase. This involves creating a suite of planning documents to help guide the team throughout the project delivery. The Planning Phase involves completing the following 10 key steps:



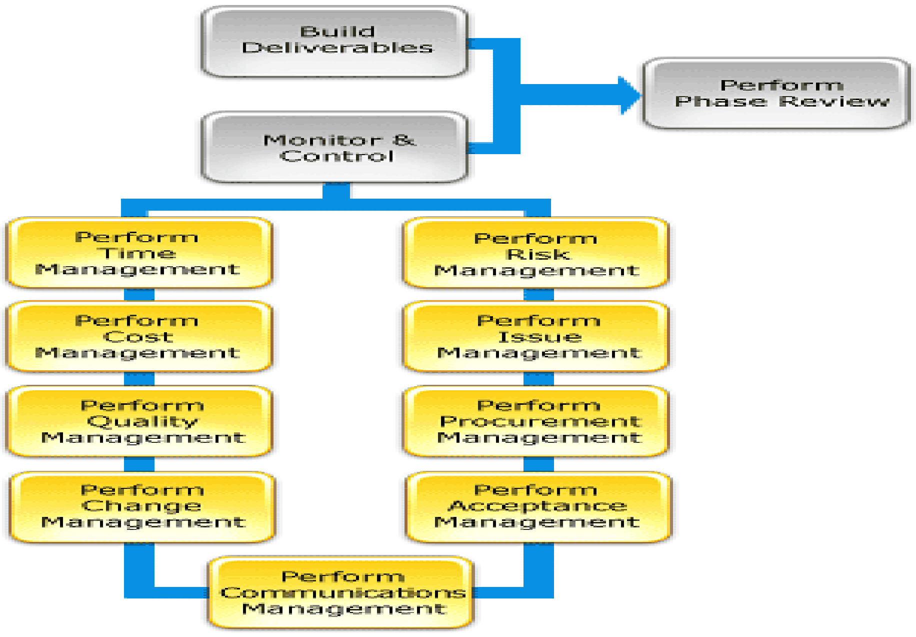
Adapted From Ncgia Application Core Curriculum- Application Issues In Gis Lecture 64

It involves setting out the roadmap for the project by creating the following plans: project plan, resource plan, financial plan, quality plan, acceptance plan and communications plan

3.5.1.3 Project Execution

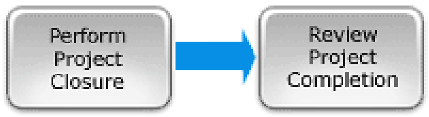
The most important issue in this phase is to ensure project activities are properly executed and controlled. During the execution phase, the planned solution is implemented to solve the problem specified in the project's requirements. In product and system development, a design resulting in a specific set of product requirements is created. This convergence is measured by prototypes, testing, and reviews. As the execution phase progresses, groups across the organization become more deeply involved in planning for the final testing, production, and support. The most common tools or methodologies used in the execution phase are an update of Risk Analysis and Score Cards, in addition to Business Plan and Milestones Reviews. With a clear definition of the project and a suite of detailed project plans, you are now ready to enter the Execution phase of the project. This is the phase in which the deliverables are physically built and presented to the customer for acceptance. While each deliverable is being constructed, a suite of management *processes* are undertaken to monitor and control the deliverables being output by the project. These processes include managing time, cost, quality, change, risks, issues, suppliers, customers and communication. Once all the deliverables have been produced and the customer has accepted the final solution, the project is ready for closure.

Adapted From Ncgia Application Core Curriculum- Application Issues In Gis Lecture 64



3.5.1.4 **Project Closure**

In this last stage, the project manager must ensure that the project is brought to its proper completion. The closure phase is characterized by a written formal project review report containing the following components: a formal acceptance of the final product by the client, Weighted Critical Measurements (matching the initial requirements specified by the client with the final delivered product), rewarding the team, a list of lessons learned, releasing project resources, and a formal project closure notification to higher management. No special tool or methodology is needed during the closure phase. It also involves releasing the final deliverables to the customer, handing over project documentation to the business, terminating supplier contracts, releasing project resources and communicating project closure to all stakeholders. The last remaining step is to undertake a Post Implementation Review to identify the level of project success and note any lessons learned for future projects.



Tutor Marked Assessment

i. How do you predict errors in software project planning
ii. Describe fully software project management life cycle

4.0 Conclusion

Software project planning consists of all the activities in building a software beginning from conception, design etc to implementation stage. It involves the various considerations such as whether the software would be in-house developed or outsourced, the budgets and monitoring etc

5.0 Summary

Software project planning involves setting out and effectively implementing the roadmap for the project by creating the following plans:

i. Project plan
ii. Resource plan
iii. Financial plan
iv. Quality plan
v. Acceptance plan
vi. Communications plan

6.0 References and Further Reading

1. Roger S. Pressman, Software Engineering (A practitioner's approach), 5th edition, 2000, Mc Graw-Hill Education
2. AgileJournal-AgileSurveyResults:SolidExperienceAndRealResults
3. AgileJournal-AgileSurveyResults:WidespreadAdoption,EmphasisonProductivityand Quality
4. Government Accountability Report (January 2003). Report GAO-03-343, National Airspace System: Better Cost Data Could Improve FAA's Management of the Standard Terminal Automation Replacement System.
5. http://www.gao.gov/cgi-bin/getrpt?GAO-03-343

SOFTWARE VALIDATION ,VERIFICATION AND TESTING

CONTENTS

1.0 INTRODUCTION

This unit provides a framework for understanding the application of software verification, validation and processes throughout the software evolution process. Typical products of this process are identified, along with their possible validation and verification objectives. The validation and verification process consists of numerous techniques and tools, often used in combination with one another. Due to the large number of validation and verification approaches in use, this unit cannot address every technique. Instead, it will analyze five categories of validation ,verification and testing approaches. These are:

i. Technical Reviews
ii. Software Testing
iii. Proof Of Correctness (Program Verification),
iv. Simulation And Prototyping, And
v. Requirements Tracing.

For each category, some representative techniques will be identified and assessed. Since the traditional focus of software validation and verification activity has been software testing, this category encompasses by far the largest number of techniques. Thus, in this unit, the software testing category will be further refined to expose the major testing techniques. This unit also addresses planning considerations for validation and verification processes, including the selection and integration of validation and verification techniques throughout the software evolution process.

2.0 Objectives

This unit is intended to support the goal of preparing professional software engineering students to:

i. Analyze the validation and verification objectives and concerns of a particular project,
ii. Examine the project's constraints,
iii. Plan a comprehensive validation and verification strategy that includes:
    a. The selection of techniques,
    b. Track the progress of the validation and verification activity,
    c. Assess the effectiveness of the techniques used and that of the overall validation and verification plan.

By end of this unit the student will be able to:

i. Define the terminology commonly utilized in the verification and validation area.
ii. Identify representative techniques for the five categories of validation and verification approaches.
iii. Explain the theoretical and practical limitations of validation and verification approaches.
iv. Describe the validation and verification objectives for typical products generated by the software evolution process.
v. Perform particular validation and verification techniques.

vi. Determine the applicability and likely effectiveness of validation and verification approaches for particular products of the software evolution process.

vii. Develop an outline for a validation and verification plan for a project that reflects understanding of validation and verification objectives, integration of techniques, problem tracking, and assessment issues.

viii. Assess the effectiveness of a validation and verification plan with respect to its objectives.

3.1 DEFINITION

Software Validation refers to the process of evaluating software at the end of its development to insure that it is free from failures and complies with its requirements. A failure is defined as incorrect product behaviour.
Often this validation occurs through the utilization of various testing approaches. Other intermediate software products may also be validated, such as the validation of a requirements description through the utilization of a prototype.

Software Verification is the process of determining whether or not the products of a given phase of a software development process fulfill the requirements established during the previous phase. Software
technical reviews represent one common approach for verifying various products. For example, a specifications review will normally attempt to verify the specifications description against a requirements description
Proof of correctness is another technique for verifying programs to formal specifications. Verification approaches attempt to identify product faults or errors, which give rise to failures.

As the complexity and diversity of software products continue to increase, the challenge to develop new and more effective validation and verification strategies continues. The validation and verification approaches that were reasonably effective on small batch-oriented products are not sufficient for concurrent, distributed, or embedded products. Thus, this area will continue to evolve as new research results emerge in response to new validation and verification challenges.
The overall objective of software Validation and Verification approaches is to insure that the product is free from failures and meets its user's expectations. There are several theoretical and practical limitations that make this objective impossible to obtain for many products.

3.1.1. Theoretical Foundations of software Validation and Verification

Some of the initial theoretical foundations for testing were presented by Goodenough and Gerhart in 1975. This theory provides definitions for reliability and validity, in an attempt to characterize the properties of a test selection strategy.
In 1983 Gourlay put forward a mathematical framework for investigating testing that enables comparisons of the power of testing methods. Howden,Beizer and Adrion summarised the theories as follows:

i. In program testing and analysis there is no general purpose testing or analysis procedure that can be used to prove program correctness.

ii. Impracticality of Testing All Data. For most programs, it is impractical to attempt to test the program with all possible inputs, due to a combinatorial explosion For those inputs selected, a testing oracle is needed to determine the correctness of the output for a particular test input.

iii. Impracticality of Testing All Paths. For most programs, it is impractical to attempt to test all execution paths through the product, due to a combinatorial explosion. It is also not possible to develop an algorithm for generating test data for paths in an arbitrary product, due to the inability to determine path feasibility.

iv. No Absolute Proof of Correctness Howden claims that there is no such thing as an absolute proof of correctness. Instead,he suggests that there are proofs of equivalency, i.e., proofs that one description of a product is equivalent to another description. Hence, unless a formal specification can be shown to be correct and, indeed, reflects exactly the user's expectations, no claim of product correctness can be made.

3.1.2. The Role of Validation and Verification in Software Evolution

The evolution of a software product can proceed in many ways, depending upon the development approach used. The development approach determines the specific intermediate products to be created. For any given project ,Validation and Verification objectives must be identified for each of the products created.

3.1.2.1 Types of Products

To simplify the discussion of V&V objectives, five types of products are considered in this unit. These types are not meant to be a partitioning of all software documents and will not be rigorously defined. Within each product type, many different representational forms are possible. Each representational form determines, to a large extent, the applicability of particular V&V approaches. The intent here is not to identify V&V approaches applicable to all products in any form, but instead to describe V&V approaches for representative forms of products.

3.1.2.2 APPROACHES

Let us briefly take look a quick look at five V&V approaches.

i. Requirements
The requirements document is customer/user-oriented. It provides an informal statement of the user's needs.

ii. Specifications
The specifications document is design oriented requirements and provides a refinement of the user's needs, which must be satisfied by the product. There are many approaches for representing specifications, both formal and informal.

iii. Designs
The product design describes how the specifications will be satisfied. Depending upon the development approach applied in the project, there may be multiple levels of designs.

iv. Implementations
Implementation normally refers to the source code for the product.

v. Changes
Changes describe modifications made to the product. Modifications are normally the result of error, corrections or additions of new capabilities to the product.

3.2 V&V Objectives

The specific V&V objectives for each product must be determined on a project-by-project basis. This determination will be influenced by the criticality of the product, its constraints, and its complexity. In general, the objective of the V&V function is to insure that the product satisfies the user needs. Thus, everything in the product's requirements and specifications must be the target of some V&V activity. In order to limit the scope of this module, however, the V&V approaches described will concentrate on the functional and performance portions of the requirements and specifications for the product. Approaches for determining whether a product satisfies its requirements and specifications with respect to safety, portability, usability, maintainability, serviceability, security, etc., although very important for many systems, will not be addressed here. This is consistent with the V&V approaches normally described in the literature. The broader picture of "assurance of software quality" is outside the scope of our work.

Limiting the scope of the V&V activities to functionality and performance, five general V&V objectives can be identified. These objectives provide a framework within which it is possible to determine the applicability of various V&V approaches and techniques.

a. Correctness: The extent to which the product is fault free.
b. Consistency: The extent to which the product is consistent
within itself and with other products.
c. Necessity: The extent to which everything in the product is necessary.
d. Sufficiency: The extent to which the product is complete.
e. Performance: The extent to which the product satisfies its performance requirements.

3.3. Software V&V Approaches and their Applicability

Software V&V activities occur throughout the evolution of the product. There are numerous techniques and tools that may be used in isolation or in combination with each other. In an effort to organize these V&V activities, five broad classifications of approaches are presented. These categories are not meant to provide a partitioning, since there are some techniques that span categories. Instead, the categories represent a practical view that reflects the way most of the V&V approaches are described in the literature and used in practice. Possible combinations of these approaches are discussed in the next section.

3.3.1 Software Technical Reviews

The software technical review process includes techniques such as walk-throughs, inspections, and audits. Most of these approaches involve a group meeting to assess a work product. A comprehensive examination of the technical review process and its effectiveness for software products is presented in all the products of the software evolution process. In particular, they are especially applicable and necessary for those products not yet in machine processable form, such as requirements or specifications written in natural language.

3.3.2 Software Testing

Software testing is the process of exercising a product to verify that it satisfies specified requirements or to identify differences between expected and actual results.

Self Assessment Exercise A

1. Define the following:

i. Software validation
ii. Software testing
iii. Software verification

2. Explain briefly the approaches used in software validation, verification and testing
3. Describe the necessary objectives of software validation, verification and testing

3.4 Levels of Testing

In this section, various levels of testing activities, each with its own specific goals, are identified and described. This listing of levels is not meant to be complete, but will illustrate the notion of
levels of testing with particular goals. Other possible levels of testing not addressed here include acceptance testing, alpha testing, beta testing, etc.

3.4.1 Module Testing

Module (or unit) testing is the lowest level of testing and involves the testing of a software module or unit. The goal of module-level testing is to insure that the component being tested conforms to its specifications and is ready to be integrated with other components of the product.

3.4.2 Integration Testing

Integration testing consists of the systematic combination and execution of product components. Multiple levels of integration testing are possible with a combination of hardware and software components at several different levels. The goal of integration testing is to insure that the interfaces between the components are correct and that the product components combine to execute the product's functionality correctly.

3.4.3 System Testing

System testing is the process of testing the integrated hardware and software system to verify that the system meets its specified requirements. Practical priorities must be established to complete this task effectively. One general priority is that system testing must concentrate more on system capabilities rather than component capabilities. This suggests that system tests concentrate on insuring the use and interaction of functions rather than testing the details of their implementations. Another priority is that testing typical situations is more important that testing special case. This suggests that test cases be constructed corresponding to high-probability user scenarios. This facilitates early detection of critical problems that would greatly disrupt a user.

There are also several key principles to adhere to during system testing:

a. System tests should be developed and performed by a group independent of the people who developed the code.
b. System test plans must be developed and inspected with the same rigor as other elements of the project.
c. System test progress must be planned and tracked similarly to other elements of the project.
d. System tests must be repeatable.

3.4.4 Regression Testing

Regression testing can be defined as the process of executing previously defined test cases on a modified program to assure that the software changes have not adversely affected the program's previously existing functions. The error-prone nature of software modification demands that regression testing be performed. Some examples of the types of errors targeted by regression testing include:

a. Data corruption errors. These errors are side effects due to shared data.
b. Inappropriate control sequencing errors. These errors are side effects due to changes in execution sequences. An example of this type of error is the attempt to remove an item from a queue before it is placed into the queue.
c. Resource contention. Examples of these types of errors are potential bottlenecks and deadlocks.
d. Performance deficiencies. These include timing and storage utilization errors.

An important regression testing strategy is to place a higher priority on testing the older capabilities of the product than on testing the new capabilities provided by the modification. This insures that capabilities the user has become dependent upon are still intact. This is especially important when we consider that a recent study found that half of all failures detected by users after a modification were failures of old capabilities, as a result of side effects of implementation of new functionality.

Regression testing strategies differ from development tests in that development tests tend to be smaller and diagnostic in nature, whereas regression tests tend to be long and complex scenarios testing many capabilities, yet possibly proving unhelpful in isolating a problem, should one be encountered. Most regression testing strategies require that some baseline of product tests be rerun. These tests must be supplemented with specific tests for the recent modifications. Strategies for testing modifications usually involve some sort of systematic execution of the modification and related areas.

At a module level, this may involve retesting module execution paths traversing the modification. At a product level, this activity may involve retesting functions that execute the modified area. The effectiveness of these strategies is highly dependent upon the utilization of test matrices (see below), which enable identification of coverage provided by particular test cases.

3.5 Testing Techniques and their Applicability

3.5.1 Functional Testing and Analysis

Functional testing develops test data based upon documents specifying the behaviour of the software. The goal of functional testing is to exercise each aspect of the software's specified behaviour over some subset of its input. Howden has developed an integrated approach to testing based upon this notion of testing each aspect of specified behaviour.

Functional testing and analysis techniques are applicable for all levels of testing. However, the level of specified behaviour to be tested will normally be at a higher level for integration and system-level testing. Thus, at a module level, it is appropriate to test boundary conditions and low-level functions, such as the correct production of a particular type of error message. At the integration and system level, the types of functions tested are normally those involving some combination of lower-level functions. Testing combinations of functions involves selection of specific sequences of inputs that may reveal sequencing errors due to:

a. Race Conditions
b. Resource Contention
c. Deadlock
d. Interrupts
e. Synchronization Issues

Functional testing and analysis techniques are effective in detecting failures during all levels of testing. They must, however, be used in
combination with other strategies to improve failure detection effectiveness.

The automation of functional testing techniques has been hampered by the informality of commonly used specification techniques. The difficulty lies in the identification of the functions to be tested. Some limited success in automating this process has been obtained for some more rigorous specification techniques.

3.5.2 Structural Testing and Analysis

Structural testing develops test data based upon the implementation of the product. Usually this testing occurs on source code. However, it is possible to do structural testing on other representations of the program's logic. Structural testing and analysis techniques include data flow anomaly detection, data flow coverage assessment, and various levels of path coverage.

Structural testing and analysis are applicable to module testing, Integration testing, and regression testing. At the system test level, structural testing is normally not applicable, due to the size of the system to be tested. For example, Petschenik FORMULAR on software analysis, a software product consisting of 1.8 million lines of code, would need over 250,000 test cases to satisfy coverage criteria. At the module level, all of the structural techniques are applicable. As the level of testing increases to the integration level, the focus of the structural techniques is on the area of interface analysis. This interface analysis may involve module interfaces, as well as interfaces to other system components.
Structural testing and analysis can also be performed on designs using manual walk-throughs or design simulations.

Structural testing and analysis techniques are very effective in detecting failures during the module and integration testing levels. Path testing catches 50% of all errors during module testing and a total of one-third of all of the errors. Structural testing is very cumbersome to perform without tools, and even with tools requires considerable effort to achieve desirable levels of coverage. Since structural testing and analysis techniques cannot detect missing functions (nor some other types of errors), they must be used in combination with other strategies to improve failure detection effectiveness.

3.5.3 Error-Oriented Testing and Analysis

Error-oriented testing and analysis techniques are those that focus on the presence or absence of errors in the programming process. Error-oriented testing and analysis techniques are, in general, applicable to all levels of testing. Some techniques, such as statistical methods, error seeding, and mutation testing, are particularly suited to application during the integration and system levels of testing.

3.5.4 Hybrid Approaches

Combinations of the functional, structural, and error-oriented techniques is called hybrid approach. These hybrid approaches involve integration of techniques, rather than their composition. Hybrid approaches, particularly those involving structural testing, are normally applicable at the module level.

3.5.5 Integration Strategies

Integration consists of the systematic combination and analysis of product components. It is assumed that the components being integrated have already been individually examined for correctness. This insures that the emphasis of the integration activity is on examining the interaction of the components. Although integration strategies are normally discussed for implementations, they are also applicable for integrating the components of any product, such as designs.

There are several types of errors targeted by integration testing:

a. Import/export range errors: This type of error occurs when the source of input parameters falls outside of the range of their destination. For example, assume module A calls module B with table pointer X. If A assumes a maximum table size of 10 and B assumes a maximum table size of 8, an import/export range error occurs. The detection of this type of error requires careful boundary-value testing of parameters.

b. Import/export type compatibility errors. This type of error is attributed to a mismatch of user-defined types. These errors are normally detected by compilers or code inspections.

c. Import/export representation errors. This type of error occurs when parameters are of the same type, but the meaning of the parameters is different in the calling and called modules. For example, assume module A passes a parameter Elapsed_Time, of type real, to module B. Module A might pass the value as seconds, while module B is assuming the value is passed as milliseconds. These types of errors are difficult to detect, although range checks and inspections provide some assistance.

d. Parameter utilization errors. Dangerous assumptions are often made concerning whether a module called will alter the information passed to it. Although support for detecting such errors is provided by some compilers, careful testing and/or inspections may be necessary to insure that values have not been unexpectedly corrupted.

e. Integration time domain/ computation errors. A domain error occurs when a specific input follows the wrong path due to an error in the control flow. A computation error exists when a specific input follows the correct path, but an error in some assignment statement causes the wrong function to be computed. Although domain and computation errors are normally addressed during module testing, the concepts apply across module boundaries. In fact, some domain and computation errors in the integrated program might be masked during integration testing if the module being integrated is assumed to be correct and is treated as a black box.

Several strategies for integration testing exist. These strategies may be used independently or in combination. The primary techniques are
top-down, bottom-up, big-bang, and threaded integration, although terminology used in the literature varies.

a. Top-down integration attempts to combine incrementally the components of the program, starting with the topmost element and simulating lower level elements with stubs. Each stub is then replaced with an actual program component as the integration process proceeds in a top-down fashion. Top-down integration is useful for those components of the program with complicated control structures. It also provides visibility into the integration process by demonstrating a potentially useful product early.

b. Bottom-up integration attempts to combine incrementally components of the program starting with those components that do not invoke other components. Test drivers must be constructed to invoke these components. As bottom-up integration proceeds, test drivers are replaced with the actual program components that perform the invocation, and new test drivers are constructed until the “top” of the program is reached. Bottom-up integration is consistent with the notion of developing software as a series of building blocks. Bottom-up integration should proceed wherever the driving control structure is not too complicated.

c. Big-bang integration is not an incremental strategy and involves combining and testing all modules at once. Except for small programs, big-bang integration is not a cost-effective technique because of the difficulty of isolating integration testing failure.

d. Threaded integration is an incremental technique that identifies major processing functions that the product is to perform and maps these functions to modules implementing them. Each processing function is called a thread. A collection of related threads is often called a build. Builds may serve as a basis for test management. To test a thread, the group of modules corresponding to the thread is combined. For those modules in the thread with interfaces to other modules not supporting the thread, stubs are used. The initial threads to be tested normally correspond to the “backbone” or “skeleton” of the product under test. (These terms are also used to refer to this type of integration strategy.) The addition of new threads for the product undergoing integration proceeds incrementally in a planned fashion.

3.5.6 Transaction Flow Analysis

Transaction flow analysis develops test data to execute sequences of tasks that correspond to a transaction, where a “transaction” is defined as a unit of work seen from a system user’s point of view. An example of a transaction for an operating system might be a request to print a file. The execution of this transaction requires several tasks, such as checking the existence of the file, validating permission to read the file, etc.

The first step of transaction flow analysis is to identify the transactions. Each transaction can then be identified as a path through the data flow diagram, with each data flow process corresponding to a task that must be tested in combination with other tasks on the transaction flow. Information about transaction flows may also be obtained from HIPO diagrams, Petri nets, or other similar system-level documentation.

Once the transaction flows have been identified, black-box testing techniques can be utilized to generate test data for selected paths through the transaction flow diagram. Some possible guidelines for selecting paths follow:

a. Test every link/decision in the transaction flow graph.
b. Test each loop with a single, double, typical, maximum, and maximumless-one number of iterations.
c. Test combinations of paths within and between transaction flows.

d. Test that the system does not do things that it is not supposed to do, by watching for unexpected sequences of paths within and between transaction flows.

Transaction flow analysis is a very effective technique for identifying errors corresponding to interface problems with functional tasks. It is most applicable to integration and system level testing. The technique is also appropriate for addressing completeness and correctness issues for requirements, specifications, and designs.

3.5.7 Stress Analysis

Stress analysis involves analyzing the behaviour of the system when its resources are saturated, in order to assess whether or not the system
will continue to satisfy its specifications. Some examples of errors targeted by stress tests include:

a. Potential race conditions
b. Errors in processing sequences
c. Errors in limits, thresholds, or controls designed to deal with overload situations
d. Resource contention and depletion

For example, one typical stress test for an operating system would be a program that requests as much memory as the system has available.

The first step in performing a stress analysis is identifying those resources that can and should be stressed. This identification is very system-dependent, but often includes resources such as file space, memory, I/O buffers, processing time, and interrupt handlers. Once these resources have been identified, test cases must be designed to stress them. These tests often require large amounts of data, for which automated support in the form of test-case generators is needed.

Although stress analysis is often viewed as one of the last tasks to be performed during system testing, it is most effective if it is applied during each of the product’s V&V activities. Many of the errors detected during a stress analysis correspond to serious design flaws.
For example, a stress analysis of a design may involve an identification of potential bottlenecks that may prevent the product from satisfying its specifications under extreme loads.

Stress analysis is a necessary complement to the previously described testing and analysis techniques for resource-critical applications.
Whereas the foregoing techniques primarily view the product under normal operating conditions, stress analysis views the product under conditions that may not have been anticipated. Stress analysis techniques can also be combined with other approaches during V&V activities to insure that the product’s specifications for such attributes as performance, safety, security, etc., are met.

3.5.8 Failure Analysis

Failure analysis is the examination of the product’s reaction to failures of hardware or software. The product’s specifications must be examined to determine precisely which types of failures must be analyzed and what the product’s reaction must be. Failure analysis is sometimes referred to as “recovery testing”. Failure analysis must be performed during each of the product’s V&V activities. It is essential during requirement and specification V&V activities that a clear statement of the product’s response to various types of failures be addressed in terms that allow analysis. The design must also be analyzed to show that the product's reaction to failures satisfies its specifications. The failure analysis of implementations often occurs during system testing. This testing may take the form of simulating hardware or software errors or actual introduction of these types of errors.

Failure analysis is essential to detecting product recovery errors. These errors can lead to lost files, lost data, duplicate transactions, etc. Failure analysis techniques can also be combined with other approaches during V&V activities to insure that the product's specifications for such attributes as performance, security, safety, usability, etc., are met.

3.5.9 Concurrency Analysis

Concurrency analysis examines the interaction of tasks being executed simultaneously within the product to insure that the overall specifications are being met. Concurrent tasks may be executed in parallel or have their execution interleaved. Concurrency analysis is sometimes referred to as "background testing".

For products with tasks that may execute in parallel, concurrency analysis must be performed during each of the product's V&V activities.
During design, concurrency analysis should be performed to identify such issues as potential contention for resources, deadlock, and priorities. A concurrency analysis for implementations normally takes place during system testing. Tests must be designed, executed, and analyzed to exploit the parallelism in the system and insure that the specifications are met.

3.5.10 Performance Analysis

The goal of performance analysis is to insure that the product meets its specified performance objectives. These objectives must be stated in measurable terms, so far as possible. Typical performance objectives relate to response time and system throughput. A performance analysis should be applied during each of the product's V&V activities. During requirement and specification V&V activities, performance objectives must be analyzed to insure completeness, feasibility, and testability.

Prototyping, simulation, or other modeling approaches may be used to insure feasibility. For designs, the performance requirements must be allocated to individual components. These components can then be
analyzed to determine if the performance requirements can be met.
Prototyping, simulation, and other modeling approaches again are techniques applicable to this task. For implementations, a performance analysis can take place during each level of testing. Test data must be carefully constructed to correspond to the scenarios for which the performance requirements were specified.

3.5.11 Proof of Correctness

Proof of correctness is a collection of techniques that apply the formality and rigor of mathematics to the task of proving the consistency between an algorithmic solution and a rigorous, complete specification of the intent of the solution. This technique is also often referred to as "formal verification." The usual proof technique follows Floyd's Method of Inductive Assertions or some Variant.

Proof of correctness techniques are normally presented in the context of verifying an implementation against a specification. The techniques are also applicable in verifying the correctness of other products, as long as they possess a formal representation.

There are several limitations to proof of correctness techniques. One limitation is the dependence of the technique upon a correct formal specification that reflects the user's needs. Current specification approaches cannot always capture these needs in a formal way, especially when product aspects such as performance, reliability, quality, etc., are considered.

Another limitation has to do with the complexity of rigorously specifying the execution behaviour of the computing environment.
For large programs, the amount of detail to handle, combined with the lack of powerful tools may make the proof technique impractical.

3.5.12 Simulation and Prototyping

Simulation and prototyping are techniques for analyzing the expected behaviour of a product. There are many approaches to constructing simulations and prototypes that are well-documented in the literature.
For V&V purposes, simulations and prototypes are normally used to analyze requirements and specifications to insure that they reflect the user's needs. Since they are executable, they offer additional insight into the completeness and correctness of these documents. Simulations and prototypes can also be used to analyze predicted product performance, especially for candidate product designs, to insure that they conform to the requirements. It is important to note that the utilization of simulation and prototyping as V&V techniques requires that the simulations and prototypes themselves be correct. Thus, the utilization of these techniques requires an additional level of V&V activity.

3.5.13 Requirements Tracing

Requirements tracing is a technique for insuring that the product, as well as the testing of the product, addresses each of its requirements. The usual approach to performing requirements tracing uses matrices. One type of matrix maps requirements to software modules. Construction and analysis of this matrix can help insure that all requirements are properly addressed by the product and that the product does not have any superfluous capabilities.

System Verification Diagrams are another way of analyzing requirements/modules traceability. Another type of matrix maps requirements to test cases. Construction and analysis of this matrix can help insure that all requirements are properly tested. It is a third type of matrix maps requirements to their evaluation approach.
The evaluation approaches may consist of various levels of testing, reviews, simulations, etc. The requirements/evaluation matrix insures that all requirements will undergo some form of V&V. Requirements tracing can be applied for all of the products of the software evolution process.

3.6 Software V&V Planning

The development of a comprehensive V&V plan is essential to the success of a project. This plan must be developed early in the project. Depending on the development approach followed, multiple levels of test plans may be developed, corresponding to various levels of V&V activities. Guidelines for the contents of system, software, build, and module test plans have been documented in the literature. These references also contain suggestions about how to document other information, such as test procedures and test cases. The formulation of an effective V&V plan requires many considerations that are defined in the remainder of this section.

3.6.1 Identification of V&V Goals

V&V goals must be identified from the requirements and specifications. These goals must address those attributes of the product that correspond to its user expectations. These goals must be achievable, taking into account both theoretical and practical limitations.

3.6.2. Selection of V&V Techniques

Once a set of V&V objectives has been identified, specific techniques must be selected for each of the project's evolving products. We consider the following approaches briefly:

3.6.2.1 Requirements

The applicable techniques for accomplishing the V&V objectives for requirements are technical reviews, prototyping, and simulation. The review process is often called a System Requirements Review (SRR). Depending upon the representation of the requirements, consistency analyzers may be used to support the SRR.

3.6.2.2 Specifications

The applicable techniques for accomplishing the V&V objectives for specifications are technical reviews, requirements tracing, prototyping, and simulation. The specifications review is sometimes combined with a review of the product's high-level design. The requirements must be traced to the specifications.

3.6.2.3 Designs

The applicable techniques for accomplishing the V&V objectives for designs are technical reviews, requirements tracing, prototyping, simulation, and proof of correctness. High-level designs that correspond to an architectural view of the product are often reviewed in a Preliminary Design Review. Detailed designs are addressed by a Critical Design Review. Depending upon the representation of the design, static analyzers may be used to assist these review processes. Requirements must be traced to modules in the architectural design; matrices can be used to facilitate this Process. Prototyping and simulation can be used to assess feasibility and adherence to performance requirements. Proofs of correctness, where applicable, are normally performed at the detailed design level.

3.6.2.4 Implementations

The applicable techniques for accomplishing the V&V objectives for implementations are technical reviews, requirements tracing, testing, and proof of correctness. Various code review techniques such as walk-throughs and inspections exist. At the source-code level, several tatic analysis techniques are available for detecting implementation errors. The requirements tracing activity is here concerned with tracing requirements to source-code modules. The bulk of the V&V activity for source code consists of testing. Multiple levels of testing are usually performed. Where applicable, proof-of-correctness techniques may be applied, usually at the module level.

3.6.2.5 Changes

Since changes describe modifications to products, the same techniques used for V&V during development may be applied during modification. Changes to implementations require regression testing.

3.6.3 Organizational Responsibilities

The organizational structure of a project is a key planning consideration for project managers. An important aspect of this structure is delegation of V&V activities to various organizations This decision is often based upon the size, complexity, and criticality of the product. In this module, four types of organizations are addressed. These organizations reflect typical strategies for partitioning tasks to achieve V&V goals for the product. It is, of course, possible to delegate these V&V activities in many other ways.

3.6.3.1 Development Organization

The development organization has responsibility for participating in technical reviews for all of the evolution products. These reviews must insure that the requirements can be traced throughout the class of products. The development organization may also construct prototypes and simulations. For code, the development organization has responsibility for preparing and executing test plans for unit and integration levels of testing. In some environments, this is referred to as Preliminary Qualification Testing. The development organization also constructs any applicable proofs of correctness at the module level.

3.6.3.2 Independent Test Organization

An independent test organization (ITO) may be established, due to the magnitude of the testing effort or the need for objectivity. An ITO enables the preparation for test activities to occur in parallel with those of development. The ITO normally participates in all of the product’s technical reviews and monitors the preliminary qualification
testing effort. The primary responsibility of the ITO is the preparation and execution of the product’s system test plan. This is sometimes referred to as the Formal Qualification Test. The plan for this must contain the equivalent of a requirements/evaluation matrix that defines the V&V approach to be applied for each requirement. If the product must be integrated with other products, this integration activity is normally the responsibility of the ITO as well.

3.6.3.3 Software Quality Assurance

Although software quality assurance may exist as a separate organization, the intent here is to identify some activities for assuring software quality that may be distributed using any of a number of organizational structures. Evaluations are the primary avenue for assuring software quality. Some typical types of evaluations to be performed where appropriate throughout the product life cycle are identified below.
Evaluation types:

i. Internal Consistency Of Product
ii. Understandability Of Product
iii. Traceability To Indicated Documents
iv. Consistency With Indicated Documents
v. Appropriate Allocation Of Sizing, Timing Resources
vi. Adequate Test Coverage Of Requirements
vii. Consistency Between Data Definitions and Use
viii. Adequacy Of Test Cases and Test Procedures
ix. Completeness Of Testing
x. Completeness Of Regression Testing

3.6.3.4 Independent V&V Contractor

An independent V&V contractor may sometimes be used to insure independent objectivity and evaluation for the customer. The scope of activities for this contractor varies, including any or all of the activities addressed for the Independent Test and Software Quality Assurance organizations.

3.7 Integrating V&V Approaches

Once a set of V&V objectives has been identified, an overall integrated V&V approach must be determined. This approach involves integration of techniques applicable to the various life cycle phases as well as delegation of these tasks among the project's organizations. The planning of this integrated V&V approach is very dependent upon the nature of the product and the process used to develop it. Traditional integrated V&V approaches have followed the "waterfall model" with various V&V functions allocated to the project's development phase. Alternatives to this approach exist, such as the Cleanroom software development process developed by IBM.

This approach is based on a software development process that produces incremental product releases, each of which undergoes a combination of formal verification and statistical testing techniques. Regardless of the approach selected, V&V progress must be tracked. Requirements/
evaluation matrices play a key role in this tracking by providing a means of insuring that each requirement of the product is addressed.

Self Assessment Test B

1. explain in details the different levels of software testing
2. discuss software integration testing with respect to its target errors
3. discuss the software testing techniques and their applications

3.8 Problem Tracking

Other critical aspects of a software V&V plan are developing a mechanism for documenting problems encountered during the V&V effort, routing problems identified to appropriate individuals for correction,
and insuring that the corrections have been performed satisfactorily. Typical information to be collected includes:

a. When the problem occurred
b. Where the problem occurred
c. State of the system before occurrence
d. Evidence of the problem
e. Actions or inputs that appear to have led to occurrence
f. Description of how the system should work; reference to relevant requirements
g. Priority for solving problem
h. Technical contact for additional information

3.9 Tracking Test Activities

The software V&V plans must provide a mechanism for tracking the testing effort. Data must be collected that enable project management to assess both the quality and the cost of testing activities. Typical data to collect include:

a. Number of tests executed
b. Number of tests remaining
c. Time used

d. Resources used
e. Number of problems found and the time spent finding them

These data can then be used to track actual test progress against scheduled progress. The tracking information is also important for future test scheduling.

3.10 Assessment

It is important that the software V&V plan provide for the ability to collect data that can be used to assess both the product and the techniques used to develop it. Often this involves careful collection of error and failure data, as well as analysis and classification of these data.

4.0 Conclusion

Software verification and validation (V&V) is an aid in determining that the software requirements are implemented correctly and completely and are traceable to system requirements. It helps to ensure that those system functions controlled by software are secure, reliable, and maintainable. Software V&V is conducted throughout the planning, development and maintenance of software systems, and may assist in assuring appropriate reuse of software.

5.0 Summary

The major objective of the software V&V process is to determine that the software performs its intended functions correctly, ensure that it performs no unintended functions, and provide information about its quality and reliability. Software V&V evaluates how well the software is meeting its technical requirements and its safety, security and reliability objectives relative to the system. It also helps to ensure that software requirements are not in conflict with any standards or requirements applicable to other system components. Software V&V tasks analyze, review, demonstrate or test all software development outputs.

The software V&V process is tightly integrated with the software development process. For each activity in software development there is a corresponding software V&V activity to verify or validate the products of those activities. This unit explains these relationships, the software V&V tasks supporting each activity, and the types of techniques that may be used to accomplish specific software V&V tasks.

6.0 Tutor Marked Assignment

1. A telecom company in Nigeria engaged as their ICT consultant. Your first task is develop their billing software plan. Fully explain how you will solve the problem.

2. Your client has an account software being developed for them. You are engaged to do the software validation, testing and verification. What does this assignment require?

7.0 References And Further Reading

i. Selby, R. W. “Combining Software Testing Strategies: An Empirical Evaluation.”

ii. Petschenik, N. H. “Practical Priorities in System *Proc. Software Testing.* Washington, D. C.: IEEE iii.Powell, P. B. “Planning for Software Validation, Verification, and Testing.” In *Software Validation*,

iv J. Edwards A Formal Approach to Software Error Removal.

v Evans, M. W. *Productive Software Test Management*.

Vi James S. Collofello. **Introduction to Software Verification and Validation**