Université de Yaoundé
Faculté des sciences
Département d'Informatique

The University of Yaounde I
Faculty of Sciences
Department of Computer Science

Software Engineering Lecture Notes INF306

2016/2017 Session

Objectives of Course

- ❖ This course teaches students the processes employed for carrying out software projects, including the design, development, testing, and deploying of practical software systems. Students are exposed to the realities involved in developing software for clients and the requirements this imposes on quality, timing of deliverables, and coordination. Students will develop hands-on experience with practical tools used in real-life applications. The course requires the completion of a group-based real-life software project.

- ❖ Provide students with an understanding of Software Architectural Design for different environments.

- ❖ Provide students with an understanding of current topics in Software Development and Software Evolution.

- ❖ Provide students with hands-on experience to work in software development project groups of real-world sizes. Students will take on practical roles and gain practical experience through projects.

Textbooks

There are a number of required textbooks for this course, but some recommended textbooks for students who wish to further explore the subject are:

I. Sommerville, L. (2010). Software Engineering, McGraw Hill (9th Edition).
II. Pressman, R. (2005). Software Engineering – A Practitioner's Approach, McGraw Hill (6th Edition).

Choosing a Project

Your project should be a web, desktop, or mobile interface. If you choose to do a mobile application, note that it must at least be possible to simulate your project on the web or the desktop, since one of your prototypes will be a simulation that you have to give to your classmates for technical evaluation.

The heart of this course is a semester-long project, in which you will design and implement any chosen project of your choice. User interface design is an iterative process, so you will build your UI not just once, but three times, as successively higher-fidelity and more complete prototypes. In order to have time for these iterations, we need to get started on the project as early as possible.

Grading:

- Every lecture will begin with a small quiz which covers the content of the previous lecture or two. This is to ascertain the level of understanding of the students. There will be approximately 4-5 small quizzes.
- Participation in lectures, TP and project group meetings will be a major factor in your grade.

Syllabus

Course Title:	**SOFTWARE ENGINEERING**
Course Code:	**INF306**
Credit Unit:	**3**

LECTURE 1: INTRODUCTION TO SOFTWARE ENGINEERING

Introduction
What is Software Engineering?
History of Software Engineering
Software Costs
The nature of software
The importance of software
Categories of software products
Software Development Environment
Attributes/characteristics of good software
Root causes of Project Success and Failure
Features of software market

LECTURE 2: PRINCIPLES OF SOFTWARE ENGINEERING

- Principle 1: Rigor and Formality
- Principle 2 - Separation of Concerns (Problems Separation)
- Principle 3 – Modularity
- Principle 4 – Abstraction
- Principle 5 - Anticipation of change
- Principle 6 – Generality
- Principle 7 - Incrementally (Incremental Development)

All these principles are explained with real life examples.

LECTURE 3: SOFTWARE PROCESS

Introduction
Software Life Cycle
Software Life Cycle Models
  Code-and-Fix Model
  Waterfall Model (Linear Sequential Model)
  The V-Shaped Model
  Incremental Process Model
  Incremental Model

Structural Testing and Functional Testing
Black Box Testing and White Box Testing

LECTURE 10: DESIGN AND IMPLEMENTATION IN THE REAL WORLD

Case study 1
Case Study 2

LECTURE 11: GROUP PROJECT PRESENTATION

For the course project, students will work in small groups to design and implement any chosen professional project of their choice a

LECTURE 1: INTRODUCTION TO SOFTWARE ENGINEERING

Introduction

Today's society is driven by speed and comfort, and software in response to societal drive is driven by ease of use and time-to-market. As a result of this, the world of software engineering continues to evolve at a rapid pace.

New releases of existing software products, as well as releases of new software products and technologies, occur often each year. The advent of new programming languages has changed our way of thinking about software and its role in contemporary society.

What is Software Engineering?

The term software engineering is composed of two words, software and engineering.

Software is more than just a program code. A program is an executable code, which serves some computational purpose. Software is considered to be a collection of executable programming code, associated libraries and documentations. Software, when made for a specific requirement is called software product.

Engineering on the other hand, is all about developing products, using well-defined, scientific principles and methods.

Software engineering is a branch of Computer Science (discipline) that uses a systematic approach (well-defined concepts and principles) to the development, implementation and maintenance of software products.

Software engineering is the application of a systematic approach to the development, implementation and maintenance of software.

Software engineering, it is the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently.

Software engineering is an engineering discipline that is concerned with all aspects of software production. Software engineers should adopt a systematic and organised approach to their work and use appropriate tools and techniques depending on the problem to be solved, the development constraints and the resources available.

The intent (essence) of software engineering is to provide a framework for building high quality software on time and within budget that meets customers' needs.

A small program can be written without using software engineering principles. But if one wants to develop a large software product, then software engineering principles are absolutely necessary to achieve high quality software. Without using software engineering principles it would be difficult to develop large programs

There is an urgent need to adopt software engineering concepts, strategies, and practices to improve the software development process in order to deliver good quality and maintainable software in time and within budget that meets customer's needs.

The context and size of the software varies in different conditions, and is influenced by the programming languages used, and the application architectures.

History of Software Engineering

(Note: Assignment 1)

Software Costs

Software costs often dominate computer system costs i.e. the costs of software on a PC are often greater than the hardware cost.

Software costs more to maintain than it does to develop. For systems with a long life, maintenance costs may be several times the development costs.

Software engineering is concerned with cost-effective software development.

The Changing Nature of Software

Today, computer software is the single most important technology on the world stage. Software is inherently flexible and can change. As requirements change through changing business circumstances, the software that supports the business must also evolve and change.

Regardless of its application domain, size or complexity, computer software will evolve over time. Change (often referred to as *software maintenance)* drives this process and occurs when errors are corrected, when the software is adapted to a new environment, when the customer requests new features or functions or when the application is reengineered to provide benefit in a modern context.

Software has become the key element in the evolution of computer-based systems and one of the most important technologies on the world stage. Over the past 50 years, software has evolved from a specialized problem solving and information analysis tool to an industry in itself. Yet we still have trouble developing high-quality software on time and within budget.

In as much as the software has been developed, it will continue to undergo some changes in order to meet the changing needs of the users. Therefore, software is dynamic not static.

Software as an entity, has affected all aspects of life. No one could have foreseen that software would become embedded in systems of all kinds: transportation, Health Care, education, telecommunications, finance, military, entertainment, etc. Despite these drastic improvements in software development (Web design software, mobile application, Artificial Intelligent software, operating systems, etc), we are yet to develop a software technology that does it all, and the likelihood of one arising in the future is slime. Therefore, software has characteristics that are considerably different than those of hardware. They include:

i. Software is an ***indispensable technology*** for business, science and engineering.
ii. Software is the ***driving force*** behind the personal computer revolution

iii. Software is ***developed***, it is not manufactured.
iv. Software does not ***"wear out":*** software is not susceptible to environmental changes that cause hardware to wear out. The life span of software can out live several generations. During its lifespan, software undergoes some changes. This nature of software aids continuity in maintenance.

The Importance of Software

Software has a dual role: It is both - a *product* and a '*vehicle'* for delivery the product.

As a software product, software have been embedded in all aspects of life- various software have been developed for different purposes e.g Medical Diagnostic software, Payroll system, Airline seat reservation, on-line registration, hotel reservation system, etc. Software enables the creation of new technologies.

As a vehicle for delivery the product, software acts as an interface between the computer user and the computer (Operating System).

NB: software products are software systems delivered to a customer with the documentation which describes how to install and use the system.

Categories of Software Products

a.) Generic Products: these are also known as Commercial Of-The-Shelf (COTS) software.

These are stand-alone systems that are marketed and sold to any customer who wishes to buy them.

COTS is built based on the software developer's specification and decision. i.e The specification of what the software should do is owned by the software developer and decisions on software change are made by the developer. E.g. compilers, OS, drivers, word processing software, spreadsheet software, database software, graphic software, file management software, game software, etc. (All these software are tailored towards developer's decisions). Therefore generic products are developed for anonymous customers. The target is generally the entire world.

Thus the advantages of COTS are:

i. It is faster to develop and provide a cheap solution
ii. Reduces or eliminates the burden of developing the software from scratch.

b.) Customized (Bespoke) Products: these are software (systems) that are developed based on customer specifications and decisions. i.e. The specification of what the software should do is owned by the customer for the software and they make decisions on software changes that are required. e.g hotel reservation system, airline seat reservation. Etc

Customized products are developed for a particular customer.

Attributes/Characteristics of Good Software

A software product can be judged by what it offers and how well it can be used. Software should deliver the require functionalities and performances to the users such as:

1. **Maintainability:** Software should be written in such a way so that it can evolve to meet the changing needs of customers
2. **Dependability/Reliability/Security:** This is a measure of how "trustworthy" the software is. Usually this is a combined measure of the safety, reliability, availability and security of a system. The software should be consistently available and users should have confidence in the software. Dependable software should not cause any physical or economic damage in the event of system failure. Software should be able to perform the basic tasks it is designed for.
3. **Efficiency:** Software should not make wasteful use of system resources such as memory and processor cycles.
4. **Portability:** Software should be usable on different platforms. It should be possible to move from one environment to another.
5. **Usability:** Software should have an appropriate user interface and documentation. Software becomes usable if it does not call for extra effort to be learnt. Usability increases with good documentation. In software operations, a lot depends on the design of the user interface and the quality of the user manual.
6. **Acceptability:** Software must be acceptable to the type of users for which it is designed. This means that it must be understandable, usable and compatible with other systems that use the software.

NB: The relative importance of these characteristics depends on the product and the environment in which it is to be used. Besides these attributes, the measure of good software is customer satisfaction. Customer satisfaction depends on the degree to which customer requirements and expectations have been met.

Features of Software Market

Considering the nature of software and the dynamics of its market, the following features are necessary:

1. **First-to-market:** the nature of the software market is such that for any product to be useful, competitive or more interesting, it must get to the market first. The software product that reaches the market first has the greatest promise for capturing a wider explosion of interest. This applies not only to new releases of new software products but also to new product features of existing software products
2. **Product Quality:** The quality of a software product remains the most dominant capability to remain relevant in the software market. Good software developers should be able to develop quality software products to stand the test of time; because test of quality is a function of test of time.
3. **Software Innovation:** This ensures continuous and competitive relevance of a product in the market. It is not over when a product is the first market, advanced features must

be continually developed and added to remain competitive and relevance. Most of the notable successes in the software domain were products of innovation.

4. **Useful partial functionality and incremental delivery:** Software developers that wait to meet all the anticipated requirements of the software products before shipment today are frustrated in the software construction market. The market is such that you get to the market as quickly as possible with what is available, interesting and useful; and bring in innovations, bug fixes, performance and reliability improvements with time, though quickly too before these lapses are exposed by rival software products or disaffection is created among the users of the products. With this, more money is made, while one maintains a competitive market position.

Root Causes of Project Success and Failure

The first step in resolving any problem is to understand the root causes. Most serious problems associated with software development are related to requirements. Three most common factors that caused software projects failure are:

i. Lack of user input
ii. Lack of understanding of customer requirements, both by the customer and the developer.
iii. Non-use of structured methodology to gather customer requirements and its analysis to arrive at the Software Requirements Specification (SRS).
iv. Incomplete requirements and specifications
v. Poor estimation of resources, effort and cost.
vi. Changing requirements and specifications

Other causes of software project failure are:

4 Unrealistic schedule or time frame
5 Inadequate staffing and resources
6 Inadequate technological skills

According to some studies, the three most important factors for software project success are:

i. User involvement
ii. Executive management support
iii. Well defined user requirement

Software Development Environment

Software development environment composes of:

- Programming Languages: Java, Python, C++, Perl, PhP, VB, …
- IDE: Visual Studio, Eclipse, NetBeans ….

- Libraries: Speech Library, Sort Library, ….
- Operating Systems: LINUX, UNIX, Windows, …
- Database: MySQL, Oracle, Database Access, …
- Version Management Tools Source Code: e.g. CVS, SVN, Git …
- Analysis and Design Tools: AlgoUML, UMBRELLO UML, Win Design, …
- Servers: Web Servers: (Apache, Microsoft's Internet Information Server (IIS), …

  Frameworks: e.g. Web Application Frameworks: (Jboss, Symphony, Zend, Django)….. Testing Frameworks: (PVS, JUnit for Java, NUnit for .NET, SAL, ICS, ACL2, HOL, etc)

N/B:

Subversion (SVN) is an open source version control system. It helps you keep track of a collection of files and folders. Any time you change, add or delete a file or folder that you manage with Subversion, you commit these changes to your Subversion repository, which creates a new revision in your repository reflecting these changes. You can always go back, look at and get the contents of previous revisions.

Some version control systems are also *software configuration management* (SCM) systems. These systems are specifically tailored to manage trees of source code and have many features that are specific to software development—such as natively understanding programming languages, or supplying tools for building software. Subversion, however, is not one of these systems. It is a general system that can be used to manage *any* collection of files.

The Concurrent Versions System (CVS), also known as the Concurrent Versioning System, is a client-server free software revision control system in the field of software development. A version control system keeps track of all work and all changes in a set of files, and allows several developers (potentially widely separated in space and time) to collaborate.

Django is a free and open source web application framework, written in Python, which follows the model–view–controller (MVC) architectural pattern

Many IDE frameworks are currently available and the most popular are:
(1) Eclipse
(2) Microsoft Visual Studio
(3) Oracle NetBeans
(4) Oracle JDevelope

Windows Mail supports the following e-mail server types:

- Post Office Protocol 3 (POP3) servers hold incoming e-mail messages until you check your e-mail, at which point they're transferred to your computer. POP3 is the most common account type for personal e-mail. Messages are typically deleted from the server when you check your e-mail.
- Internet Message Access Protocol (IMAP) servers let you work with e-mail messages without downloading them to your computer first. You can preview, delete, and organize messages directly on the e-mail server, and copies are stored on the server until you choose to delete them. IMAP is commonly used for business e-mail accounts.
- Simple Mail Transfer Protocol (SMTP) servers handle the sending of your e-mail messages to the Internet. The SMTP server handles outgoing e-mail, and is used in conjunction with a POP3 or IMAP incoming e-mail server.

ASSIGNMENT ONE

1. History of Software Engineering

LECTURE 2: PRINCIPLES OF SOFTWARE ENGINEERING

What does it take to ensure a successful software development project?

Seven Basic Principles of Software Engineering

The seven basic principles of Software Engineering are:

- Principle 1 - Rigor and formality
- Principle 2 - Separation of Concerns (Problems Separation)
- Principle 3 – Modularity
- Principle 4 – Abstraction
- Principle 5 - Anticipation of change
- Principle 6 – Generality
- Principle 7 - Incrementally (Incremental Development

These principles guide the development of all aspects of software development.

Principle 1 - Rigor and formality:
Software development is a creative process but it must be practiced systematically. Creativity often leads to imprecision and inaccuracy. Rigor and formality Increase the confidence in the creative results

Rigor is a necessary complement to creativity that increases our confidence in our developments

Rigor helps to:
...produce more reliable products
...control cost
...increase confidentiality in products

- Formality is "rigor -- mathematically sound" Often used for mission critical systems. Formality is rigor at the highest degree

Situations where rigor and formality are applied:

- Algorithm design and analysis
- Mathematical (formal) analysis of program correctness
- Rigorous documentation of the software development process.

Example: Elevator Management System

- Define requirements

    - must be able to carry up to 400 Kg. (safety alarm and no operation if overloaded)
    - emergency brakes must be able to stop elevator within 1 m. and 2 sec. in case of cable failures
  - Later, verify their fulfillment

Principle 2 - Separation of Concerns (Problems Separation): This is a design principle for separating a computer program into distinct sections, such that each section addresses a separate concern. The idea behind this modularization, allows the programmer to reduce the complexity of the system being designed.

The principle of separation is one way to conquer program complexity (Divide and Conquer).

The separation-of-concerns principle is one of the essential principles in software engineering. It addresses one issue at a time. It says that software should be decomposed in such a way that different "concerns" or aspects of the problem at hand are solved in well-separated modules or parts of the software.

Some dimension of separation: Cost, System Performance (product quality) and Time
Advantage:

- The value of separation of concerns is simplifying development and maintenance of computer programs. When concerns are well-separated, individual sections can be reused, as well as developed and updated independently.
- Separation of concerns helps to Divide a problem into parts that can be dealt with separately.
- Separation of concerns helps to control the complexity of programs.

In a nutshell, it promotes the separation of different interests in a problem, solving them separately without requiring detailed knowledge of the other parts, and finally combining them into one result.
Separate functionality from efficiency
Separate Requirements specification from design
Separate non-functional requirements from functional requirements.

Example:

Hypertext Markup Language (HTML), Cascading Style Sheets (CSS), and JavaScript (JS) are complementary languages used in the development of webpages and websites. HTML is mainly used for organization of webpage content, CSS is used for definition of content presentation style, and JS defines how the content interacts and behaves with the user. Historically, this was not the case though. Prior to the introduction of CSS, HTML performed both duties of defining semantics and style

Principle 3 – Modularity: The principle of modularity is a specialization of the principle of separation of concerns. Following the principle of modularity implies separating software into components according to functionality and responsibility.
Modularity is the application of separation of concern.
A modular system consists of well defined, manageable units with well-defined interfaces among the units.
A system is considered modular if it consists of discreet components so that each component can be implemented separately and a change to one component has minimal impact on other components.

Advantage: It enhances design clarity, which in turn eases implementation, debugging, testing, documenting and maintenance of the software product.

Principle 4– Abstraction:
The principle of abstraction is another specialization of the principle of separation of concerns. Following the principle of abstraction implies separating the behavior of software components from their implementation. It requires learning to look at software and software components from two points of view: what it does, and how it does it. Failure to separate behavior from implementation is a common cause of unnecessary coupling.

Abstraction is a special case of separation of concerns.

Abstraction hides details and provides simplified description of a system.
One important concept of abstraction is information hiding (data encapsulation).

The type of abstraction to apply depends on purpose:

For Example: The GUI of an Employee Attendance System abstracts from the system the aspect of <u>Signing In,</u> other abstractions are needed to support <u>cumulative hours,</u> <u>Extract hours/Over time,</u> etc.

Advantage: Hides details and provides simplified description of system.

Principle 5 - Anticipation of change:

Change is inevitable in any software development process. Not anticipating change often leads to high cost and unmanageable software. Software development deals with inherently changing requirements.

- Ability to support software evolution requires anticipating potential future changes
- It is the basis for software evolvability.

Advantages:
Anticipation of change helps to:
...create a software infrastructure that absorbs changes easily
...enhance reusability of components
...control/save cost in the long run

Principle 6 - Generality:

While solving a problem, try to discover if it is an instance of a more general problem whose solution can be reused in other cases. In every problem, attempt to find a more general solution. General problem is often easier to solve. A generalized solution may be reusable
Not generalizing often leads to continuous redevelopment of similar solutions. Software development involves building many similar kinds of software (components). Software developers cannot tolerate building the same thing over and over again.
Advantages:
Generality leads to...
...increased reusability
...increased reliability
...faster development
...reduced cost

Principle 7 - Incrementally (Incremental Development):

Process proceeds in a stepwise fashion (*increments*)

Delivering a large product as a whole, and in one shot, often leads to dissatisfaction and a product that is “not quite right”.

Deliver the first prototype and then incrementally add effort to turn prototype into product

Incrementality leads to...
...the development of better products
...early identification of problems
...an increase in customer satisfaction
…Active involvement of customer

Concluding Remarks:

- These fundamental principles guide all aspects of software development.
- Remember

  - Tools, methodologies, process models and techniques will evolve, the principles remain the same

LECTURE 3: SOFTWARE PROCESS

Introduction

Software process is a set of activities (a sequence of overlapping activities) involve in the development of a software product.

In any software development project, we have the Client, Developer and User

SOFTWARE LIFE CYCLE

Software Life Cycle or Software Development Life Cycle (SDLC) is a set of activities (a sequence of overlapping activities) involve in the development of a software product.

The life cycle defines a methodology for improving the quality of software and the overall development process.

Software development activities (software life-cycle activities) include the following:

(1) Requirement Determination/Specification

Determining software requirements- i.e. identifying the features needed in a new software system is usually an evolutionary process. The software developer must study and understands the current (existing) system by carrying out feasibility study. There must be a clear definition of the problem statement.

An important task in creating/developing software is extracting the requirement through problem definition.

Once the general requirements are gathered from the clients and stakeholders, an analysis of the scope and objectives of the software system should be clearly stated. (Therefore, in requirement definition, the following should be clearly stated:

- Problem statement
- Determination of scope
- Determination of objectives, alternatives and constraints.

At this stage, i.e. software specification, a comprehensive description of the intended purpose of the intended software should be state.

The complete specification of a system is thus a very difficult task. Poor software specification is one the most common sources of project failure.

Requirement errors are the most common type of systems development error and the most expensive errors to fix. (Because it involves re-specification, redesign and re-implementation, retesting, etc).

In view of this, the objective of studying the users' requirement or carrying out feasibility study is to determine whether the request is feasible (worth-while) before a recommendation is reached to do nothing, improve or modify the existing system, or to develop a new system from scratch.

In order to determine what functionalities the software should contain, proper data gathering (fact finding techniques) should be used.

Conclusively, at this stage the functional requirements are identified for the new system.

(2) System Design

The design phase focuses on how the software requirement specification can be realized.

The software developer may decide to design a new algorithm or used an existing algorithm with modifications.

He may decide to develop software from scratch (customized software) and enhance it (commercial off-the-shelf software) and enhance its capabilities.

System design can be a logical (architectural) design or a physical design.

The logical design is perform using standard design modeling tools such as DFD, ERD, UML, Data Dictionary, Decision Trees and Decision Tables, etc.

The physical design is the actual software developed.

Modular approach of software development can be used to achieve a better design.

Two main attributes of design are:

i. Coupling
ii. Cohesion

COUPLING: Coupling is the manner and degree of interdependency between software modules (subsystems) i.e. the relationship between software modules – the manner in which software modules communicate. Lower coupling is better because if two subsystems are loosely coupled, they are relatively independent, and thus, modifications on one subsystem will have little impact on the other.

If the two subsystems are strongly coupled, modification on one subsystem is likely to have impact on the other.

COHESION: Cohesion is the manner and degree to which tasks performed by a single software module are related to one another – a measure of the degree to which the elements of a module are functionally related i.e. the degree to which the entities and their functions within a module are related to one another. Higher cohesion is better i.e. the strength of dependency within a subsystem.

Tightly coupled systems tend to exhibit the following developmental characteristics which are often seen as disadvantages:

i. A change in one module usually forces a ripple effect (multiplicative effect) of changes in other modules.
ii. Assembly or integration of modules might require more effort and/or time due to the increased inter-module dependency.
iii. A particular module might be harder to reuse and/or test because dependent modules must be included.

NB: Coupling and Cohesion are the two terms which frequently occur together during design. Together, they talk about the quality a module should have. Coupling talks about the interdependencies between the various modules while cohesion describes how functions within a module are related. System developers should strive for low coupling and high cohesion.

(3) System Development

This phase produces the actual code that will be delivered to the customer as a working system. The developer may either decide to develop customized software based on client's needs or COTS based on developer's needs.

(4) System Implementation and Testing

Before a system is put into operation, its components (program modules) must be tested to make sure that they function individually (as a unit) and as a whole.

Testing phase is crucial for ensuring the quality of the software product.

Testing is usually done in stages: First as Unit Testing (i.e. testing individual components) then as System Testing or Integrated Testing (testing integrated components) using test data which can be live or artificial data. Live test data are those that are actually extracted from organization files while artificial test data (generated data) are created solely for test purposes.

Following system testing is acceptance testing (testing to satisfy the purchaser) based on user's specifications. i.e testing with customer data to check that the system meets the customer's need.

System implementation means to put a new or enhanced system into operation.

Implementation includes all those activities that take place to convert from the old system to the new system. Implementing a system, whether a new one or an enhanced system, consists of three primary activities, they include:

- Training
- Conversion:
  - Direct approach

    - Parallel approach
    - Phased approach
    - Pilot approach
- Post implementation review

The post implementation review (Evaluation) provides first-hand source of information for maintenance.

The objectives of the post implementation review are to:

i. Determine whether the systems goals and objectives have been achieved.
ii. Determine whether personnel procedures and operating activities have improved.
iii. Determine whether user requirement have been et while simultaneously reducing errors and costs.
iv. Determine whether known or unexpected limitations of the system need attention.

(5) System Maintenance

System maintenance means updating and improving the software to ensure continuous usefulness.

After the software is installed, it must be maintained, meaning that the software must be modified and kept up-to-date for continuous relevance.

Maintenance is performed for two reasons:

i. The first is to correct software errors. No matter how thoroughly the system is tested, bugs (errors) may creep into the programs.
ii. The second reason for performing system maintenance is to enhance the software capabilities in response to changing organizational needs.

SOFTWARE LIFE CYCLE MODELS (SOFTWARE PROCESS MODELS)

Software Life Cycle is a period within which software is effective and yielding required results

The traditional methods of software development were mainly characterized by:

i. Wrong, incomplete and unstable functional system requirements
ii. Unspecified design strategies
iii. Incorrect implementation
iv. Poor testing plan
v. Inadequate maintenance strategies
vi. Complex documentation, etc.

Because of the inadequacies of the traditional methods of software development, Software Development Process was introduced which ensures:

i. The management of software life cycle (consisting of phases)
ii. Which phases are required during system development?
iii. The order of the phases.
iv. What happens in each phase
v. The outcome of each phase.

Plan-driven and agile processes

- ✧ Plan-driven processes are processes where all of the process activities are planned in advance and progress is measured against this plan.
- ✧ In agile processes, planning is incremental and it is easier to change the process to reflect changing customer requirements.
- ✧ In practice, most practical processes include elements of both plan-driven and agile approaches.
- ✧ There are no right or wrong software processes.

DEFINITION: A software life cycle model is a standardized format for planning, organizing and running a new development project.

Some examples of software life cycle models include:

i. Code-and-Fix Model
ii. Waterfall Model
iii. Incremental Model
iv. Rapid Application Development (RAD) Model
v. Spiral Model
vi. Prototyping Model
vii. Object-Oriented Model
viii. Fountain Model

SOFTWARE LIFE CYCLE MODELS

A number of different process models for software engineering have been proposed, but all define a set of framework activities for building software.

The framework of activities are organized into a process-flow, they may be linear, incremental, or evolutionary. The terminology and details of each process model differ, but the generic framework activities remain reasonably consistent.

Most system development *Process Models* in use today have evolved from three primary approaches: *Ad-hoc Development*, *Waterfall Model*, and the *Iterative* process (Incremental and Evolutionary).

The importance of software and the need for improvement in many facets of its development has led to practical and research interest in improving the management of the software development process. A number of software process models have been discussed in the literature. However, relatively little management research has examined the software development process, either as a unique task or in the context of new product development.

Code and Fix Model

Early systems development often took place in a rather chaotic and haphazard manner, relying entirely on the skills and experience of the individual staff members performing the work. Today, many organizations still practice *Ad-hoc Development* either entirely or for a certain subset of their development (e.g. small projects).

The Software Engineering Institute at Carnegie Mellon University points out that with *Ad-hoc Process Models*, "process capability is unpredictable because the software process is constantly changed or modified as the work progresses. Schedules, budgets, functionality, and product quality are generally (inconsistent). Performance depends on the capabilities of individuals and varies with their innate skills, knowledge, and motivations.

In the code and fix model, a product is developed without specifications or any attempt at design. Instead, the developer simply builds a product that is reworked as many times as necessary to satisfy the client.
This is an ad-hoc (unplanned) approach and not well defined. Basically it is a simple two-phase model. The first phase is to write code and the next phase is to fix it as shown in Figure 2.1. Fixing in this content may be error correction or addition of further functionality.
Although this approach may work well on small programming exercise 100 or 2000 lines long, this model is total unsatisfactory for software of any reasonable size.
The cost of the development using this approach is actually very high as compared to the cost of a properly specified and carefully designed product. In addition, maintenance of the product can be extremely difficult without specification or design documents.

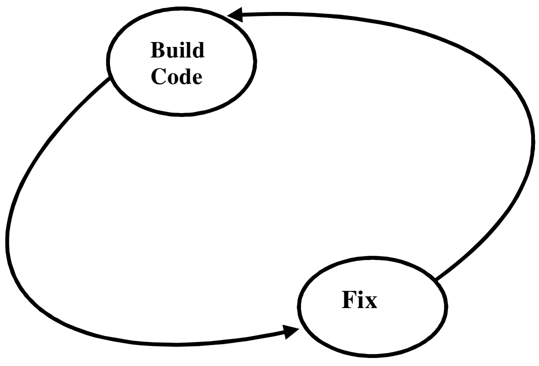


Figure 3.1: Build and Fix Model

The Waterfall Model

The waterfall model is the oldest among the software development models introduced by Royce in 1970. The waterfall model is also known as linear sequential model. The waterfall model is the classical life-cycle model that describes a systematic sequence of activities in a software life-cycle. The waterfall model is applicable when users' requirements are well-defined and reasonably stable.

This model has five phases: Requirement Determination/ Specification, System Design, System Development, System Implementation and Testing, and System Maintenance.

Because of the cascade from one phase to another, this model is known as the waterfall model.

The result of each waterfall activity provides feedback to earlier phases. In waterfall model, after phase is finished, the developers' proceeds to the next phase. In view of this, the waterfall provides no room for revisit previous phases.

Each of the activities in the waterfall model provides feedback to developers responsible for earlier activities.

Waterfall model discourages revisiting of previous phases once it is complete. This "inflexibility" in the waterfall model has been a source of criticism by supporters of other more "flexible" models.

Some problems of waterfall model are:

i. It is difficult to define all requirements at the beginning of a project
ii. This model is not suitable for accommodating any change
iii. A working version of the system is not seen until late in the project's life.

Therefore the customer must have patience but at times, users may not be patience to get the final product.

Due to these weaknesses, the application of waterfall model should be limited to situations where the users' requirements and their implementation are well understood. E.g. some accounting systems.

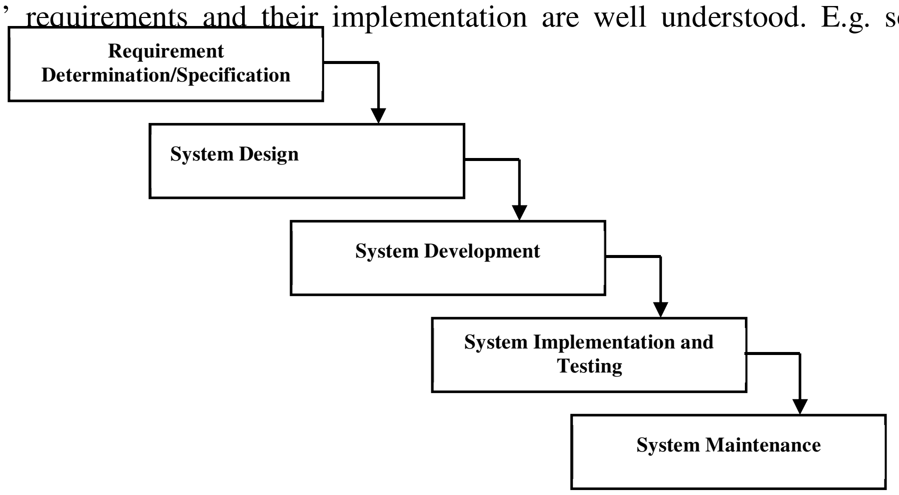

Requirement Determination/Specification
System Design
System Development
System Implementation and Testing
System Maintenance

Figure 3.2: Waterfall Model of Software Development (Royce, 1970)

The V-Shaped Model

The V-shape model is in substance identical to the waterfall model, but is unique because it helps individuals conceptualize changes in the degree of detail involved in activities throughout the software development process. The V-shaped model may be considered as extension of the waterfall model. The V-shape model was originally presented by R.B. Jensen and C.C. Tonies in 1979 and is further discussed by R.B. Rowen (1990). Just like the waterfall model, the V-Shaped life cycle is a sequential path of execution of processes. Instead of moving in a linear way, the process steps are bent upwards after the coding phase to form the typical V shape as shown in Figure 2.3. Each phase must be completed before the next phase begins. Testing is emphasized in this model more than the waterfall model. The V-shaped model demonstrates the relationships between each phase of the development life cycle and its associated phase.
As the early development phases proceed towards coding, the level of abstraction in the project decreases as specifications become increasingly detailed. The coding phase entails the highest degree of detail in the project. During the testing phase, the level of abstraction again increases as development moves from the component level to the subsystem level and finally to the system level. The final step is the acceptance test, which signifies approval of the product by the customer or a customer representative. In the V-shape model, correspondence between the requirements and design phases and the subsequent testing levels is apparent. This model provides a different view of software development by depicting the level of detail inherent in the project at any given time, but the shortcomings of the underlying waterfall model still hold. The V-shape model depicts a linear and controlled process, and implicitly assumes that the customer will understand the content and activities of all software development tasks and subtasks (even the most detailed. This model has same drawback like the waterfall model.

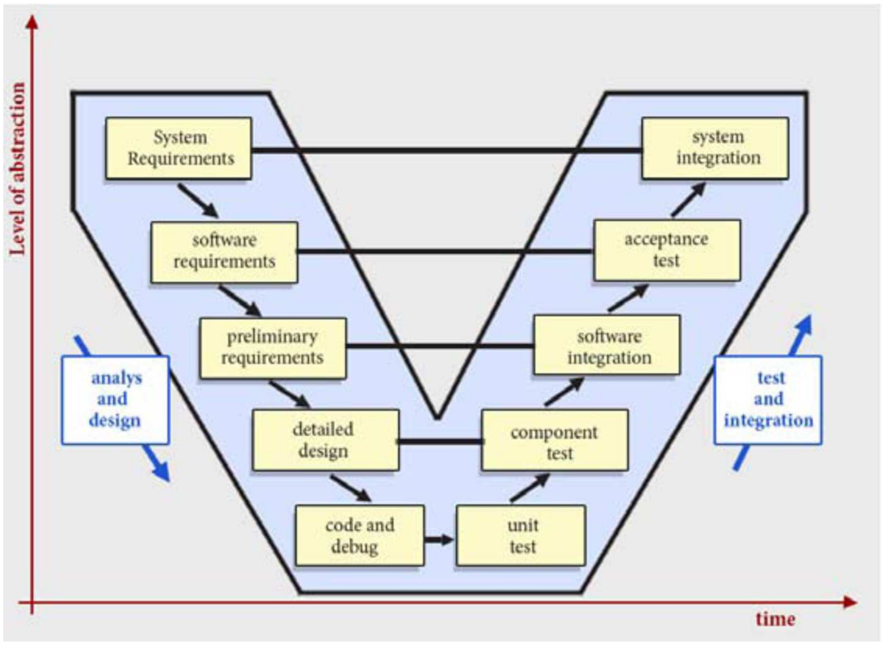


Figure 3.3: V-Shaped Model

❖ INCREMENTAL PROCESS MODEL

There are many situations in which the initial software requirements are reasonably well-defined. Furthermore, there may be a compelling need to provide a limited set of software functionalities to users quickly, refine and expand on that functionality in later software releases. In such cases a process model that is designed to produce the software in increments is the best option.

(A) The Incremental Model

This model has the same phases as the waterfall model but in an iterative fashion i.e. the model is conducted in several cycles.

A useable product is released at the end of each cycle, with each release providing additional functionality

During the first requirement analysis phase, users and developers specify as many requirements as possible and prepare a Software Requirement Specifications (SRS) document.

Developers and users then prioritize these requirements. Developers implement the specified requirements in one or more cycles of the design, implementation and testing, based on the defined priorities.

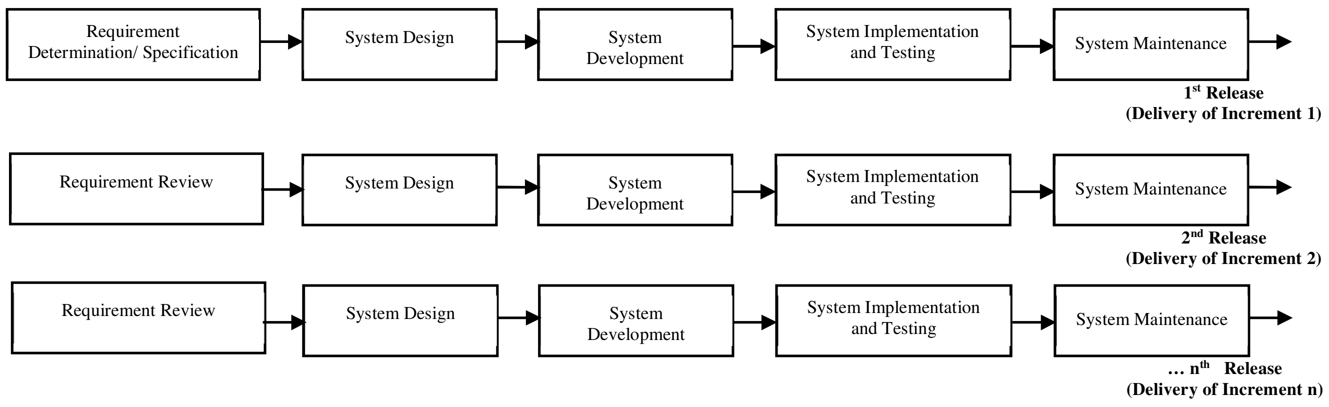

Requirement Determination/ Specification
System Design
System Development
System Implementation and Testing
System Maintenance
1st Release (Delivery of Increment 1)
Requirement Review
System Design
System Development
System Implementation and Testing
System Maintenance
2nd Release (Delivery of Increment 2)
Requirement Review
System Design
System Development
System Implementation and Testing
System Maintenance
... nth Release (Delivery of Increment n)

Figure 3.4 Incremental Process Model

The incremental model, like prototyping and other evolutionary models is iterative in nature. But unlike prototyping, the incremental model focuses on the delivery of an operational product with each increment.

Early increments are "lower" version of the final product but they also provide a platform for evaluation by the user. E.g. word processing software developed using the incremental model might deliver basic file management, editing and document products functions in the first release; more sophisticated editing and document production capabilities in the second release, spelling and grammar checking in the third release, and advanced page layout capabilities in the fourth release and so on.

This model encourages partial functionality.

This model ensures full user participation in the software development particularly after the first release.

The aim of the waterfall and prototyping models is the delivery of a complete, operational and good quality product. In contrast, this model does deliver an operational quality satisfies only a subset of the customer's requirement.

(B) The Rapid Application Development (RAD) Model

The RAD (Rapid Application Development) model is based on prototyping and iterative development with no specific planning involved.

Rapid Application Development (RAD) is an incremental process model that emphasizes a short development cycle. It was proposed by IBM in the 1980s.

The RAD model is a "high-speed" adaptation of the waterfall model, in which rapid development is achieved by using a component-based constructs approach as shown in Figure 2.5.

If requirement are well understood and project scope is constrained, the RAD process enables a development team to create a "fully functional system" within a very short time period.

In this model, **user involvement** is essential from requirement phase to delivery of the product.

The continuous user participation ensures the involvement of user's expectations and perspective in requirements elicitation analysis and design of the system. A rapid prototype is first build and is given to user for evaluation. The user feedback is obtained and prototype is refined. The process continues, till the requirements are finalized.

In this model, multiple software teams work in parallel on different systems. As a result, quick initial views about the product are possible. Therefore, development time of the product is reduced.

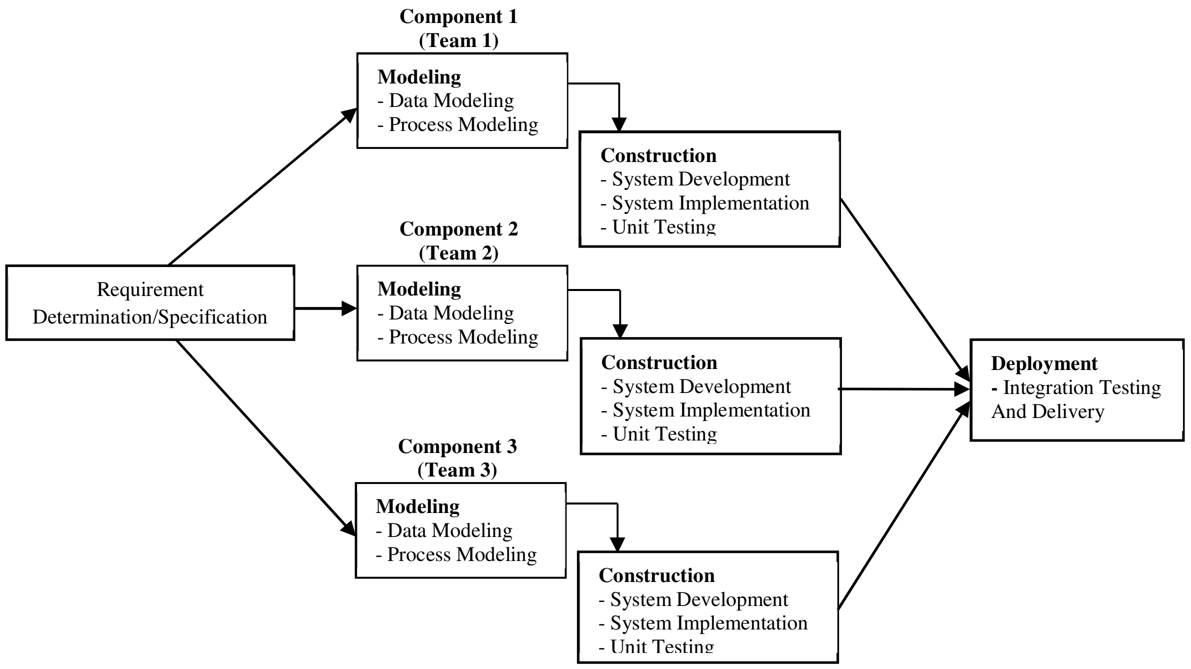


Figure 3.5: RAD Model

If users cannot be involved throughout the life cycle, this may not be an appropriate model.

Due to the fact that, many software projects have poorly defined requirements at the start, prototyping, RAD or other evolutionary approaches are much better process options.

Like all process models RAD approach has drawbacks:

1. For large, but scalable projects, RAD requires sufficient human resources to create the right number of RAD teams.
2. Highly specialized and skilled developers are expected and such developers may not available very easily.
3. It may not be effective, if system cannot be properly modularized; because the components necessary for RAD will be problematic.

❖ **Evolutionary Process Models**

Software, like all complex systems, evolves over a period of time. Users' requirements often change as development proceeds; making a straight-line path to an end product unrealistic but a limited version must be introduced to meet competitive or business pressure. In such a situation, software engineers need a process model that can accommodate a product that evolves over time. Evolutionary models are iterative in nature.

(A) Spiral Model

The problem with the traditional software process model is that they do not deal sufficiently with uncertainty which is inherent to software projects. Important software projects have failed because project risks were neglected and nobody was prepared when something unforeseen happened. Barry Boehm recognized this and tried to incorporate the "project risk" factor into a life cycle model. The result is the spiral model.

The spiral model, originally proposed (developed) by Boehm in 1988, is an evolutionary software process model that combines (couples) the iterative nature of <u>prototyping model</u> with the systematic nature of the <u>waterfall model</u> making it a <u>hybrid model</u>.

The spiral model is a risk-driven process model that is used to guide multi-stakeholders and software engineers of software intensive project.

It has three main distinguishing features:

(1) It is a cyclic approach for incrementally growing (developing) a system while decreasing its risk.
(2) It is Risk-driven: Risk Analysis includes identifying, estimating, and monitoring technical feasibility and management risks, such as schedule slippage and cost overrun. After testing the build, at the end of first iteration, the customer evaluates the software and provides feedback.
(3) It is a future-driven approach that accommodates changes in the life cycle thereby ensuring feasible and mutually satisfactory system solutions.

Using the spiral model, software is developed in a series of evolutionary releases. During early iterations, the release might be a paper model or prototype. During later iterations increasingly more complete versions (releases) of the engineered system are produced.

The spiral model is divided into a set of framework activities defined by the software engineering team.

It has four main activities and a loop of the spiral from the X-axis clockwise through 360-degrees represents one phase.

The spiral model is visualized as a process passing through some number of iterations. With a four quadrant diagram representing the following four activities:

1. Requirement Determination/Specification: Determination of scope, objectives, alternative solutions and constraints
2. Risk Analysis: Evaluate (analyze) alternatives and attempts to identify and resolve the risks involved i.e. major sources of risks are identified and resolve to an extent – Risk evaluation of customer's requirements.

3. System Development and Implementation: Product development, implementation and testing. Develop and verify next-level product.
4. Assessment: Customer interactions and evaluation. Plan next phases of the cycle

The Boehm Spiral Model involving four iterations is shown in the diagram in Figure 2.6

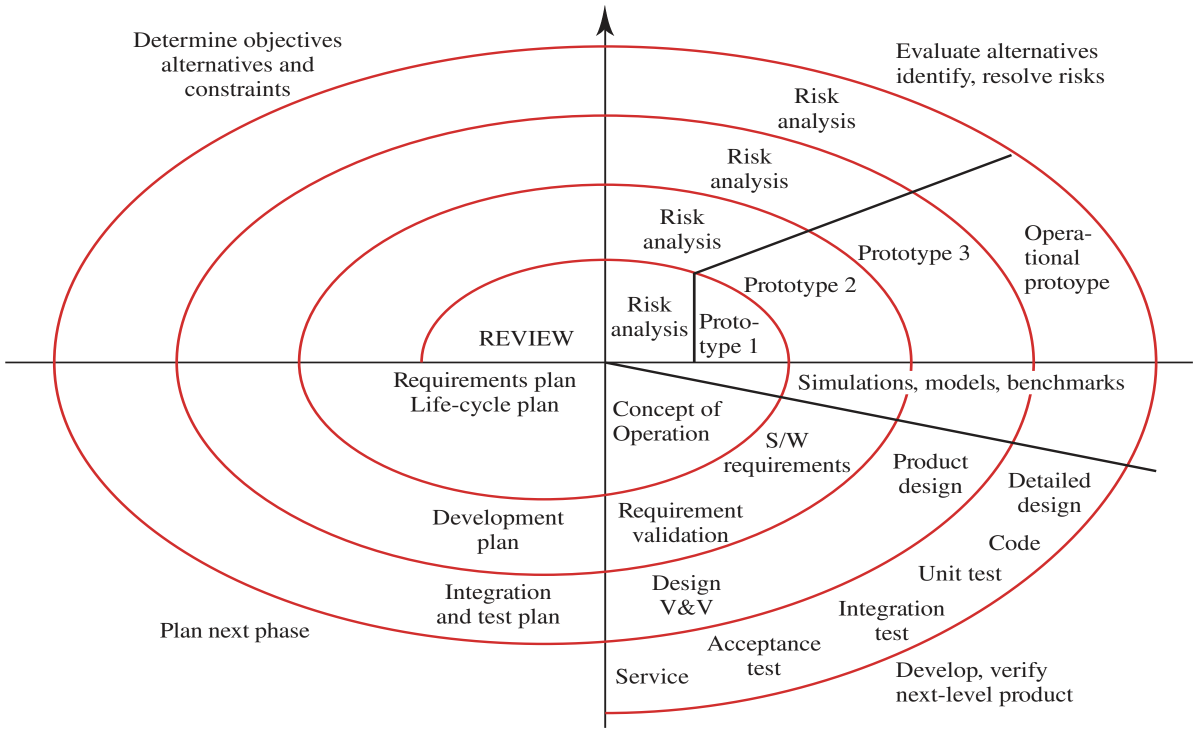


Figure 3.6: Spiral Model (Boehm, 1988)

Based on the customer evaluation, software development process enters into the next iteration and subsequently follows the linear approach to implement the feedback suggested by the customer. The process of iteration s along the spiral continues throughout the life of the software

The model is an iterative model with Risk Management at regular stages in the development of the product.

Since end-user requirements are hard to define at the initial stage (recall that determining the system requirements is usually an evolutionary process throughout the system life-cycle), it is therefore natural to develop software in an experimental fashion.

An iterative and interactive development of software is made possible by gaining experience with the behavior and shortcomings of product prototypes. In the iterative development process, the risk of the software being unacceptable to the user is much reduced.

Iterative processes are preferred by some developers because it allows a potential of reaching the design objectives (goals) of a customer who does not know how to define what he wants. As a result of this, spiral model has greater flexibility than waterfall model. It has good project visibility.

This model is recommended when users' requirement are not well-defined.

The major disadvantages of this model are: it needs technical expertise in risk analysis; it is cost intensive; and needs more time.

Prototyping Model

This model is also known as rapid prototyping model.

The prototyping approach to software development focuses on producing software products quickly.

Prototyping is seen as a means of reducing risk, discovering potential problems before developing a full-fledged system.

Prototyping can allow developers to rapidly construct early or primitive versions or test versions or limited versions of software products that users can evaluate. Users' evaluations can then be incorporated as feedback to refine the emerging system specifications.

The initial prototype may be a usable program, but is not suitable as the final software product. The reason may be poor performance. The code for the prototype may be thrown away or reused; however, the experienced gathered from developing the prototype helps in developing the actual system. Therefore, the development of a prototype might involve extra cost.

Software prototypes have come in different forms including throw-away prototypes, demonstration systems, quick-and-dirty prototypes and evolutionary prototypes (Note: throw-away prototype because original codes are thrown away and revolutionary prototypes because original codes are modified to accommodate new requirements and reused).

The sole use of this model is to determine the customer' real needs, therefore prototyping technologies help users to define the functional requirement of the software.

After the finalization of the software requirement and specification (SRS) document, the prototype is discarded and the actual system is then developed using the Waterfall Approach. Thus, it is used as an input to waterfall model and produces maintainable and good quality software. Therefore, the development cost is expensive. Prototyping is iterated to obtain feedback from clients and changes in system concepts. Thus the complete working system can be developed through a continual revising/refining the inputs specifications. This has the advantage of always providing a working version of the emerging system, while redefining functional users' requirements, design and testing activities.

The evolutionary prototype may solve the waiting problem in the waterfall model (it is not necessary to wait until the end of development cycle for a working version of the software). Figure 2.7 shows the diagram of the prototype model.

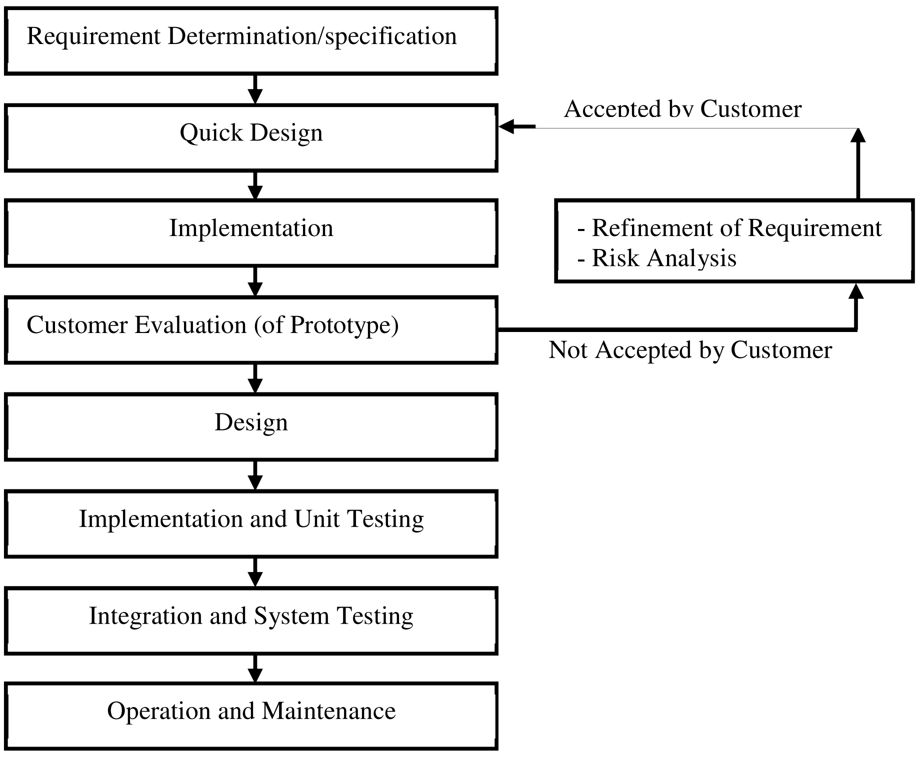

Requirement Determination/specification
Quick Design
Accepted by Customer
Implementation
- Refinement of Requirement
- Risk Analysis
Customer Evaluation (of Prototype)
Not Accepted by Customer
Design
Implementation and Unit Testing
Integration and System Testing
Operation and Maintenance

Figure 3.7: Prototyping Model

Selection Criteria for Appropriate Model

The selection of a suitable model is based on the following characterises/categories:

(i) Users' Requirements
(ii) Development Team
(iii) Users' Involvement
(iv) Project type and associated risk.

NB: GENERAL REMARK

In summary, where the requirements are easy to establish and are reasonably stable; the development is customer specific; and changes are not foreseen in the near future, the development of software through the "Waterfall Model" is recommended. (Is this an ideal case? Certainly NO)

Where requirements are difficult to establish in clear terms, the "Waterfall Model" becomes very ineffective; and once the software requirements are not clear, system design and development become very costly operations. When the software engineer is confronted with such a scenario, the development of the software is carried out through an iterative (incremental) process modelling or an evolutionary process modelling.

SRS document is a large document, written in a natural language and contains a description of what the system will do without describing how it will be done.

LECTURE 4: SOFTWARE REQUIREMENT ENGINEERING (REQUIREMENTS ANALYSIS AND SPECIFICATION)

Introduction

Once we have established the feature set and have gained agreement with the customer, we can move on to defining the more specific requirements that will be needed to provide the proposed solution.

Before we start to develop our software, it becomes quite essential for us to understand and document the exact requirement of the customer. Experienced members of the development team carry out this job. They are called as system analysts.

Basic Definition

What is Software Requirement?

Software requirement can be defined can be defined as:

- *A software capability needed by user to solve a problem in order to achieve an objective.*
- *A software capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed documentation.*

What is Software Requirement Engineering?

- *Requirements engineering refers to the process of defining, documenting and maintaining requirements.*

What is Requirement Engineering Process?

The activities involved in requirements engineering vary widely, depending on the type of system being developed and the specific practices of the organization(s) involved. These may include

- **Requirement Elicitation:** The process of collecting the requirements of a system from users, customers and other stakeholders.The practice is also sometimes referred to as "requirement gathering"
- **Requirement Analysis:** checking requirements and resolving stakeholder conflicts
- **Requirement Specification:** documenting the requirements in a requirements document (Requirement Specification Document)
- **Requirement Design:** deriving models (Architectural design) of the system
- **Requirement Validation:** checking that the documented requirements and models are consistent and meet stakeholder needs
- **Requirement Management:** managing changes to the requirements as the system is developed and put into use.

The Five Steps in Problem Analysis

Problem Analysis can be defined as

The process of understanding real-world problems and users' needs and proposing solutions to meet those needs.

In order to be able to do problem analysis, it would be helpful to define what a problem is.

A problem can be defined as the difference between things as perceived and things as derived.

The goal of problem analysis is to gain a better understanding before development begins of the problem to be solved. The specific steps that must be taken in order to achieve the goal are:

1. Gain agreement on the problem definition.
2. Understanding the root causes – the problem behind the problem.
3. Identify the stakeholders and the users.
4. Define the solution system boundary.
5. Identify the constraints to be imposed on the solution.

Step 1: Gain Agreement on the Problem Definition

The first step is to gain agreement on the definition of the problem to be solved. One of the simplest ways to gain this agreement is to simply write the problem down and see whether everyone agrees.

As part of this process, it is often beneficial to understand some of the benefits of a proposed solution, being careful to make sure that the benefits are described in the terms provided by the customers/users. Having the user describe the benefits provides additional contextual background on the real problem. In seeing the benefits from the customer's point of view, we also gain a better understanding of the stakeholder's view of the problem itself.

Problem statement format is presented in Table 3.1

Table 3.1: Problem statement format

Elements	**Description**
The problem of	Describe the problem
Affects	Identify stakeholders and users affected by the problem
The result of which	Describe the impact of this problem on stakeholders and business activity.
Benefits	Indicate the proposed solution and list a few key benefits

Step 2: Understanding the Root Causes – The Problem Behind the Problem

The software developer must gain an understanding of the real problem and its real causes by determining what factors contributed to the problem.

Step 3: Identify the stakeholders and the users.

Effectively solving any complex problem typically involves satisfying the needs of a divers group of stakeholders. Stakeholders (decision makers and users) will typically have varying perspectives on the problem.

Step 4: Define the solution system boundary.

Once the problem statement is agreed on and the users and stakeholders are identified, we can turn our attention to defining a system that can be deployed to address the problem. In doing so, we enter an important transition state wherein we have to keep two things in mind: an understanding of the problem and the considerations of a potential solution.

The next important step is to determine the boundaries of the solution system. The system boundary (limits of the system) defines the borders between the solution and the real world that surround the solution. In other words, the system boundary describes an envelope in which the solution system in contained.

Step 5: Identify the constraints to be imposed on the solution

We'll define a constraint as:

A restriction on the degree of freedom we have in providing a solution.

Each constraint has the potential to severely restrict our ability to deliver a solution as we envisioned. Therefore, each constraint must be carefully considered as part of the planning process and many may even cause us to reconsider the technological approach we have initially envisioned. Some of the constraints might be time constraint, financial constraint: system constraint (is the solution to be built on an existing system? What operating system and environments must be supported), etc.

Functional and Non-Functional Requirements

Functional requirements are *requirements that describe the functionalities of a system.* They depend on the type of software, expected users, system boundaries and the type of system where the software is used.

Functional system requirement should describe the system services in detail.

Examples of Functional Requirements are

1. The user shall be able to search either all of the initial set of databases or select a subset from it.

2. The system shall provide appropriate viewers for the users to read documents in the document store.
3. Every order shall be allocated a unique identifier (ORDER_ID) which the user shall be able to copy to the account's permanent storage area.
4. The system shall automatically generate student ID, Customer ID, staff ID, Patient ID, etc.
5. The system reaches its optimum point when patterns start repeating or when a chaotic situation is observed or when a cyclic fashion is observed.

Other Examples:

Example 1- Consider the case of the Library Management System

F1: Search Book function

Input: An author's name

Output: Details of the author's books and the location of these books in the library

So the function Search Book (F1) takes the author's name and transforms it into book details. Functional requirements actually describe a set of high-level requirements, where each high-level requirement takes some data from the user and provides some data to the user as an output. Also each high-level requirement might consist of several other functions.

Example 2- Consider the case of an ATM (Automated Teller Machine) System

F1: Cash Withdraw function

Input: The Amount

Output: The withdraw cash function first determines the type of account that the user has and the account number from which the user wishes to withdraw cash. It checks the balance to determine whether the requested amount is available in the account. If enough balance is available, it outputs the required cash; otherwise it generates an error message.

Non-Functional requirements deal with the characteristics of the system which cannot be expressed as functions - such as the maintainability of the system, portability of the system, usability of the system, etc (Overall system performance)

Non-Functional requirements are constraints on the services or functions offered by the system such as time constraint, budget constraint (unrealistic budget), constraints on the development process, standards, etc. (System Constraints).

Therefore, non-functional requirements refers to system performance and system constraints.

❖ The important parts of Software Requirement Specification (SRS) document are:

- Functional requirements of the system
- Non-functional requirements of the system, and
- Goals of implementation

Requirements Validation

Requirements validation is concerned with demonstrating that the requirements define the system that the customer really wants. Requirement error costs are high, so validation is very important. Fixing requirement errors after delivery may cost up to 100 times the cost of fixing implementing errors.

Requirement Checking

- **i.** **Validity:** Does the system provide the functions which best support the customer's needs?
- **ii.** **Consistency:** Are there any requirements conflict?
- **iii.** **Completeness:** Are all functions required by the customer included?
- **iv.** **Realism:** Can the requirement be implemented giving the available budget and technology?
- **v.** **Verifiability:** Can the requirements be checked?

PROFESSIONAL PROJECT

PRACTICAL 1: ASSIGNMENT TWO

(1a) Define functional and non-functional requirements. Using any project below, identify the functional and non-functional requirements

a. Orphanage Information system
b. Library Management System
c. Airline Seat Reservation System
d. Computerized Banking System
e. Computerized Billing System
f. Result Processing System

(1b) with referenced to your chosen project, recommend an appropriate software process model with concrete reasons.

(1c) what do you understand by the term capability model (CMM)
(Discuss explicitly)

(1d) Depict the logical view of your chosen project using:

a. ERD
b. DFD

(e) Differentiate between Top-down and Bottom-Up modeling approach.

N/B: You must carry out your feasibility study

LECTURE 5: SOFTWARE DESIGN

Introduction

Software design is a creative process because it deals with the development of the actual mechanics for a new workable system. The design must be detail as it is the basis for programming and System Implementation. Design is a problem-solving activity and as such, very much a matter of trial and error.
Design is a highly significant phase in the software development where the software plans "how" a software system should be produced in order to make it functional, reliable and reasonably easy to understand, modify and maintain.
Software Requirement Specification (SRS) is an input to the design process and tells us "how" a software system works.

Why is Design important?

A good design is the key to successful software product. A well-designed system is easy to implement, understand and reliable and allows for smooth evolution. Without design, we risk building an unstable system:

- One that will fail when small changes are made.
- One that will be difficult to maintain
- One whose quality cannot be assessed until late in the software process.

Therefore, software design should contain a sufficiently complete, accurate and precise solution to a problem in order to ensure its good implementation.

There are three **characteristics** that serve as a guide for the evolution of a good design:

i. The design must implement all of the explicit requirements contained in the analysis model and it must accommodate all of the implicit requirements desired by the customer.
ii. The design must be readable, understandable guide for those who generate code and for those who test and subsequently support the software.
iii. The design should provide a complete picture of the software, addressing the data, functional and behavioral domain from an implementation perpective.

What is Software Architecture?

Software architecture describes the entities of a system as seen by the programmer. i.e the conceptual and functional behaviour of a system as seen by the programmer. E.g Data-oriented architecture, process- oriented architecture, and object- oriented architecture.

There are three types of design: Logical (low Fidelity Prototype), Conceptual design and Physical design.

- Logical design is the design on paper (low Fidelity Prototype)

- Conceptual or structural design of the system while Physical design is the actual software. Conceptual or structural design of the system is independent of the implementation language and platform while physical design is based on the actual implementation language and platform that will be used.

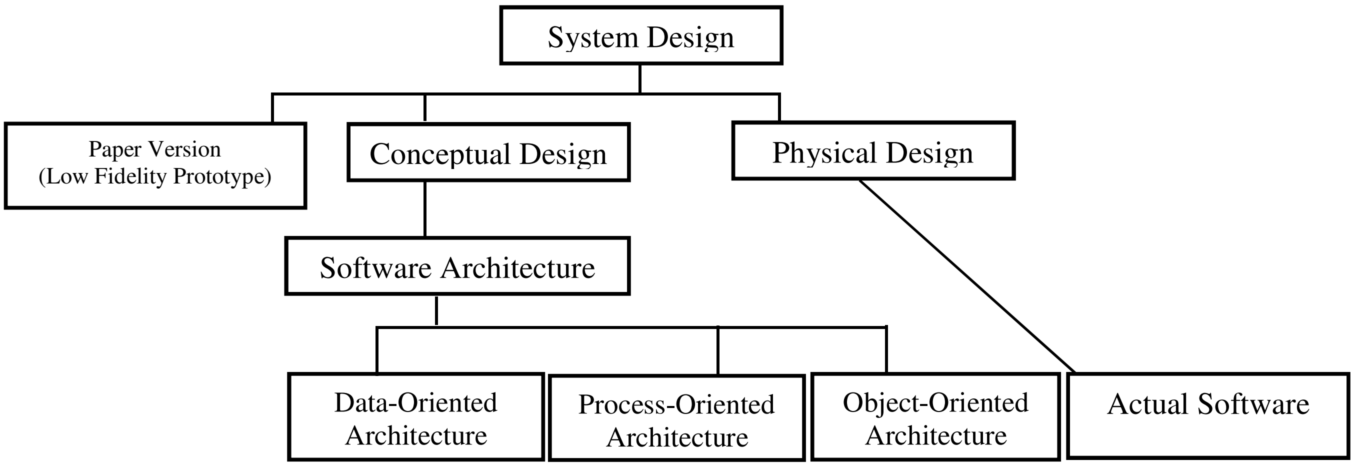

System Design
Paper Version (Low Fidelity Prototype)
Conceptual Design
Physical Design
Software Architecture
Data-Oriented Architecture
Process-Oriented Architecture
Object-Oriented Architecture
Actual Software

Figure 6.1: Graphical Representation of System Design

Process-Oriented Architecture: e.g Data Flow Diagram

Data-Oriented Architecture: e.g Entity Relationship Diagram

Object-Oriented Architecture: e.g Unified Modeling Language

Attributes of Good Design	
Attributes	**Description**
User Familiarity	The interface should use terms and concepts which are drawn from the experience of the people who will make most use of the system. (Match between system and the real world: The system should speak the **users' language,** with words, phrases and concepts familiar to the user, rather than system-oriented terms. Follow real-world conventions, making information appear in a natural and logical order.)
Consistency and Standards	The interface should be consistent in that, wherever possible, comparable operations should be activated in the same way**.** (Users should not wonder whether different words, situations, or actions mean the same thing. Follow platform conventions; Strive for consistency)
Minimal Surprise	Users should never be surprised by the behaviour of a system.
Recoverability	The interface should include mechanisms to allow users to recover from errors. (User control and freedom: Users often choose system functions by mistake and will need a clearly marked "emergency exit" to leave the unwanted state without having to go through an extended dialogue. Support undo and redo; Permit Easy Reversal of Actions)
User Guidance	The interface should provide meaningful feedback when errors occur and provide context-sensitive user help facilities.
User Diversity	The interface should provide appropriate interaction facilities for different types of system user. (Cater for Universal usability)

Other Attributes of a Good Design

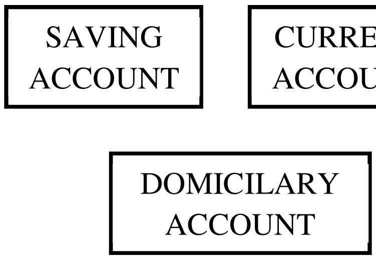
There are three main attributes of a good design. They are; **Modularity, Cohesion** and **Coupling**

MODULARITY: A modular system consists of well defined, manageable units with well defined interfaces among the units. Modular approach of software development can be used to achieve a better design.

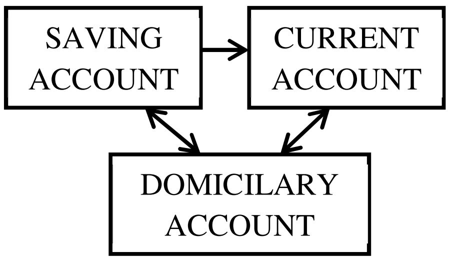
COUPLING: Coupling is the manner and degree of interdependency between software modules (subsystems) i.e. the relationship between software modules – the manner in which software modules communicate. Lower coupling is better because if two subsystems are loosely coupled, they are relatively independent, and thus, modifications on one subsystem will have little impact on the other.

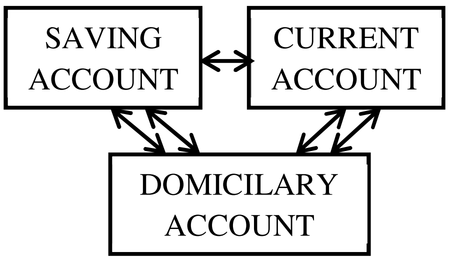
If the two subsystems are strongly coupled, modification on one subsystem is likely to have impact on the other.

e.g. Assuming the following are subsystems SAVING ACCOUNT, CURRENT ACCOUNT, and DOMICILARY ACCOUNT, the degree of coupling between them can be indicated as shown below:



(a) Uncoupled: No Dependencies





(b) Loosely Uncoupled: Some Dependencies **(c)Highly Coupled: Many Dependencies**

COHESION: Cohesion is the manner and degree to which tasks performed by a single software module are related to one another – a measure of the degree to which the elements of a module are functionally related i.e. the degree to which the entities and their functions within a module are related to one another. Higher cohesion is better i.e. the strength of dependency within a subsystem.

Design Strategies

A good system design strategy is to organize the program modules in such a way that are easy to develop and maintain. There are many strategies or techniques for performing system design. They include bottom up approach, top down approach and hybrid approach.

❖ **Bottom-up Design**

Bottom-up design begins the design with the lowest level modules or subsystems, and progresses upward to the main program, module or subsystem.

A bottom-up approach is the piecing together of systems to give rise to more complex systems, thus making the original systems sub-systems of the emergent system. In a bottom-up approach the individual base elements of the system are first specified in great detail. These elements are then linked together to form larger subsystems, which then in turn are linked, sometimes in many levels, until a complete top-level system is formed.

❖ **Top-down Design**

Top down design as the name implies takes a high level definition of the problem and subdivides it into sub problems, which can then be solved to a piece that will be easy to code.

Top-down design begins the design with main or top-level module, and progresses downwards to the lowest level modules or subsystem. i.e. a top down design starts by identifying the major modules of the system, decomposing them into their lower level modules and iterating until the desired level of detail is achieved. This stepwise refinement; starting from an abstract design, in each step the design is refined to a more concrete level, until we reach a level where no more refinement is needed and the design can be implemented directly. Most design methodologies are based on this approach and this is suitable if the specifications are clear and development is from scratch.

A top-down approach (also known as stepwise design and in some cases used as a synonym of *decomposition*) is essentially the breaking down of a system to gain insight into its compositional sub-systems. In a top-down approach an overview of the system is formulated, specifying but not detailing any first-level subsystems. Each subsystem is then refined in yet greater detail, sometimes in many additional subsystem levels, until the entire specification is reduced to base elements.

❖ **Hybrid Design**

Pure bottom-up and pure top-down approaches are often not practical. Hybrid approach combines bottom-up approach and top-down approaches. Hybrid approach has really become popular after the acceptance of reusability of modules.

Assignment: Differentiate between top-down approach and bottom-up design

Answer:

Top-down Approach	**Bottom-down Approach**
Top-down approach proceeds from the abstract entity to get the concrete design.	Bottom-up approach proceeds from the concrete design to get the abstract design.
Top-down design is mostly used in designing brand new systems.	Bottom-up design is sometimes used when reverse engineering a design. i.e when one is trying to figure out what somebody else design in an existing system.
Top-down design begins the design with the main to top-level module and progresses downwards to the lowest level modules or subsystems.	Bottom –up design begins with the lowest module or subsystem and progresses upwards to the main program module or subsystem.

Real life sometimes is a combination of bottom-up design and top-down design. For instance m most modelling sessions tend to be iterative in nature bouncing back and forth between top-down and bottom-up models as the need arises.

Implementation of Architectural Design

(Use any Programming Language of your choice)

Please refer to the last practical assignment.

LECTURE 6: OBJECT MODELLING USING UML

Introduction to UML

Model

A model captures aspects important for some application while omitting (or abstracting) the rest. A model in the context of software development can be graphical, textual, mathematical, or program code-based. Models are very useful in documenting the design and analysis results. Models also facilitate the analysis and design procedures themselves. Graphical models are very popular because they are easy to understand and construct. UML is primarily a graphical modelling tool. However, it often requires text explanations to accompany the graphical models.

Need for a model

An important reason behind constructing a model is that it helps manage complexity. Once models of a system have been constructed, these can be used for a variety of purposes during software development, including the following:

- Analysis
- Specification
- Code generation
- Design
- Visualize and understand the problem and the working of a system
- Testing, etc.

In all these applications, the UML models can not only be used to document the results but also to arrive at the results themselves. Since a model can be used for a variety of purposes, it is reasonable to expect that the model would vary depending on the purpose for which it is being constructed. For example, a model developed for initial analysis and specification should be very different from the one used for design. A model that is being used for analysis and specification would not show any of the design decisions that would be made later on during the design stage. On the other hand, a model used for design purposes should capture all the design decisions. Therefore, it is a good idea to explicitly mention the purpose for which a model has been developed, along with the model.

UML, as the name implies, is a modeling language. It may be used to visualize, specify, construct, and document the artifacts of a software system. It provides a set of notations (e.g. rectangles, lines, ellipses, etc.) to create a visual model of the system. Like any other language, UML has its own syntax (symbols and sentence formation rules) and semantics (meanings of symbols and sentences). Also, we should clearly understand that UML is not a system design or development methodology, but can be used to document object-oriented and analysis results obtained using some methodology.

Origin of UML

In the late 1980s and early 1990s, there was a proliferation of object-oriented design techniques and notations. Different software development houses were using different notations to document their object-oriented designs. These diverse notations used to give rise to a lot of confusion. UML was developed to standardize the large number of object-oriented modelling notations that existed and were used extensively in the early 1990s. The principles ones in use were:

- Object Management Technology [Rumbaugh, 1991]
- Booch's methodology [Booch, 1991]
- Object-Oriented Software Engineering [Jacobson, 1992]
- Odell's methodology [Odell, 1992]
- Shaler and Mellor methodology [Shaler and Mellor, 1992]

It is needless to say that UML has borrowed many concepts from these modeling techniques. Especially, concepts from the first three methodologies have been heavily drawn upon. UML was adopted by Object Management Group (OMG) as a de facto standard in 1997. OMG is an association of industries which tries to facilitate early formation of standards.

We shall see that UML contains an extensive set of notations and suggests construction of many types of diagrams. It has successfully been used to model both large and small problems. The elegance of UML, its adoption by OMG, and a strong industry backing have helped UML find widespread acceptance. UML is now being used in a large number of software development projects worldwide.

UML Diagrams

UML can be used to construct nine different types of diagrams to capture five different views of a system. Just as a building can be modeled from several views (or perspectives) such as ventilation perspective, electrical perspective, lighting perspective, heating perspective, etc.; the different UML diagrams provide different perspectives of the software system to be developed and facilitate a comprehensive understanding of the system. Such models can be refined to get the actual implementation of the system.

The UML diagrams can capture the following five views of a system:

- User's view
- Structural view
- Behavioral view
- Implementation view
- Environmental view

❖ **User's view:** This view defines the functionalities (facilities) made available by the system to its users. The user's view captures the external users' view of the system in terms of the functionalities offered by the system. The users' view is a black-box view of the system where the internal structure, the dynamic behavior of different system components, the implementation etc. are not visible. The users' view is very different from all other views in the sense that it is a functional model compared to the object model of all other views. The users' view can be considered as the central view and all other views are expected to conform to this view. This thinking is in fact the crux of any user centric development style.

❖ **Structural view:** The structural view defines the kinds of objects (classes) important to the understanding of the working of a system and to its implementation. It also captures the relationships among the classes (objects). The structural model is also called the **static model**, since the structure of a system does not change with time.

❖ **Behavioral view:** The behavioral view captures how objects interact with each other to realize the system behavior. The system behavior captures the time-dependent (dynamic) behavior of the system.

❖ **Implementation view**: This view captures the important components of the system and their dependencies.

❖ **Environmental view:** This view models how the different components are implemented on different pieces of hardware.

USE CASE DIAGRAM

Use Case Model

The use case model for any system consists of a set of "use cases". Intuitively, use-cases represent the different ways in which a system can be used by the users. A simple way to find all the use cases of a system is to ask the question: "What the users can do using the system?" Thus for the Library Information System (LIS), the use cases could be:

- Issue-book
- Query-book
- Return-book
- Create-member
- Add-book, etc

Use cases correspond to the high-level functional requirements. The use cases partition the system behavior into transactions, such that each transaction performs some useful action from the user's point of view. To complete each transaction may involve either a single message or multiple message exchanges between the user and the system to complete.

Uses cases model all the external entities of a system called actors.

Representation of Use Cases

Use cases can be represented by drawing a use case diagram and writing an accompanying text elaborating the drawing. In the use case diagram, each use case is represented by an ellipse with the name of the use case written inside the ellipse. All the ellipses (i.e. use cases) of a system are enclosed within a rectangle which represents the system boundary. The name of the system being modelled (such as Library Information System) appears inside the rectangle.

LECTURE 7: OBJECT MODELLING USING UML CONT'D

LECTURE 8: SOFTWARE DESIGN CONT'D

Implementing Architectural Design (Using any Programming Language of your choice)

LECTURE 9: SOFTWARE TESTING

Introduction

Software Testing is an empirical (experimental) investigation conducted to provide stakeholders with information about the quality of the product or service under test.
System testing is executing a program to check its functionality with the view of finding errors or defects.
- Two types of errors can be checked: syntax errors and logical errors. A syntax error is a program statement that violates one or more rules of the language in which it is written (language of implementation). A logical error on the other hand deals with incorrect data fields, out-of-range items, and invalid combination.
- Testing is essential to the success of a system. In system testing, performance and acceptance standards are developed.

What is Software Testing?

Testing is the process of evaluating a system or its component(s) with the intent to find that whether it satisfies the specified requirements or not.

Testing is the process of executing a program with the intent of finding errors.

This activity results in the actual, expected and difference between their results. In simple words testing is executing a system in order to identify any gaps, errors or missing requirements in contrary to the actual desire or requirements

Testing is the most important part of the Software Development Life Cycle (SDLC). One cannot release the final product without passing it through the testing process.

The purpose of software testing is to find bugs/defects in the software. Software testing is an essential part of Software Engineering.

Goals of Software Testing

The general aim of testing is to affirm the quality of software systems by systematically exercising the software in carefully controlled circumstances to:

- Reveal faults (finding errors: syntax errors and logical errors)
- Establish confidence in the software
- Clarification of user's specification
-

Challenges of Software Testing

- Testing is huge cost of product development
- Incomplete, informal and changing specifications or incomplete users' requirements.
- Lack of Software testing tools
- Testing effectiveness and software quality is hard to measure
- ---

Limitation of software testing

Software testing can only shows the presence of errors not their absence

Software Testing Quality Attributes

- ❖ Fault tolerance
  - Programmatically cause a fault and test that the system can recover
- ❖ Security
  - How secured is the software product?
- ❖ Usability
  - Is the software user-friendly, Measure user performance on some task
- ❖ Portability
  - Test against multiple platforms
- ❖ Evolvability
  - Design extension

Our aim is to design effective test cases in order to find maximum possible errors. The more we test, the more we are confident about our design.

Differences between Verification & Validation

These two terms are very confusing for people, who use them interchangeably. Let's discuss about them briefly.

Verification	**Validation**
Are you building it right?	Are you buildinq the riqht thing?
Ensure that the software system meets all the functionality.	Ensure that functionalities meet the intended behaviour.
Verification takes place first and includes the checking for documentation, code etc.	Validation occurs after verification and mainly involves the checking of the overall product.
Done by developers.	Done by Testers.
Have static activities as it includes the reviews, walkthroughs, and inspections to verify that software is correct or not.	Have dynamic activities as it includes executing the software against the requirements.
It is an objective process and no subjective decision should be needed to verify the Software.	It is a subjective process and involves subjective decisions on how well the Software works.

Difference between Testing, Quality Assurance and Quality Control

Most people are confused with the concepts and difference between Quality Assurance, Quality Control and Testing. Although they are interrelated and at some level they can be considered as the same activities, but there is indeed a difference between them. Mentioned below are the definitions and differences between them:

Quality Assurance	**Quality Control**	**Testing**
Activities which ensure the implementation of processes, procedures and standards in context to verification of developed software and intended requirements.	Activities which ensure the verification of developed software with respect to documented (or not in some cases) requirements.	Activities which ensure the identification of bugs/error/defects in the Software.

Focuses on processes and procedures rather then conducting actual testing on the system.	Focuses on actual testing by executing Software with intend to identify bug/defect through implementation of procedures and process.	Focuses on actual testing.
Process oriented activities.	Product oriented activities.	Product oriented activities.
Preventive activities	It is a corrective process.	It is a preventive process
It is a subset of Software Test Life Cycle (STLC)	QC can be considered as the subset of Quality Assurance.	Testing is the subset of Quality Control.

Difference between Testing and Debugging

Testing: It involves the identification of bug/error/defect in the software without correcting it. Normally professionals with a Quality Assurance background are involved in the identification of bugs. Testing is performed in the testing phase.

Debugging: It involves identifying, isolating and fixing the problems/bug. Developers who code the software conduct debugging upon encountering an error in the code. Debugging is the part of White box or Unit Testing. Debugging can be performed in the development phase while conducting Unit Testing or in phases while fixing the reported bugs.

Types of Software Testing

To prove to the customer that the software has reached the desired level of quality, all applicable aspects of the program needs to be tested. However, testing all possible per-mutations of a program would take too long, and may not be economically feasible. Using only one type of test will probably not be sufficient to achieve the requested level of quality, mainly because some tests are good at exposing certain types of error, but bad at others. There are many types of tests to choose from, all with their distinctive strengths and weaknesses. When the system seems to work from the developer's point of view, one should investigate if this is true from the user's point of view.

A successful test should uncover new errors, not "prove" that there are no errors, as the former adds more value to the program.

This section describes the different types of testing which may be used to test a Software during SDLC. Some of them include:

(1) White Box Testing
(2) Black Box Testing
(3) System Testing
(4) Acceptance Testing
(5) Performance Testing
(6) Regression Testing

❖ **White Box Testing and Black Box Testing**

White Box Testing (also known as clear, glass box or structural testing) is a testing technique which evaluates the code and internal structure of the program.

White-box testing is a verification technique software engineers can use to examine if their code works as expected.
White-box testing is testing that takes into account the internal mechanism of a system or component

The connotations of "clear box", "glass box" or "structural testing" appropriately indicate that you have full visibility of the internal workings of the software product, specifically, the logic and the structure of the code.

With white-box testing, you must run the code with predetermined input and check to make sure that the code produces predetermined outputs.

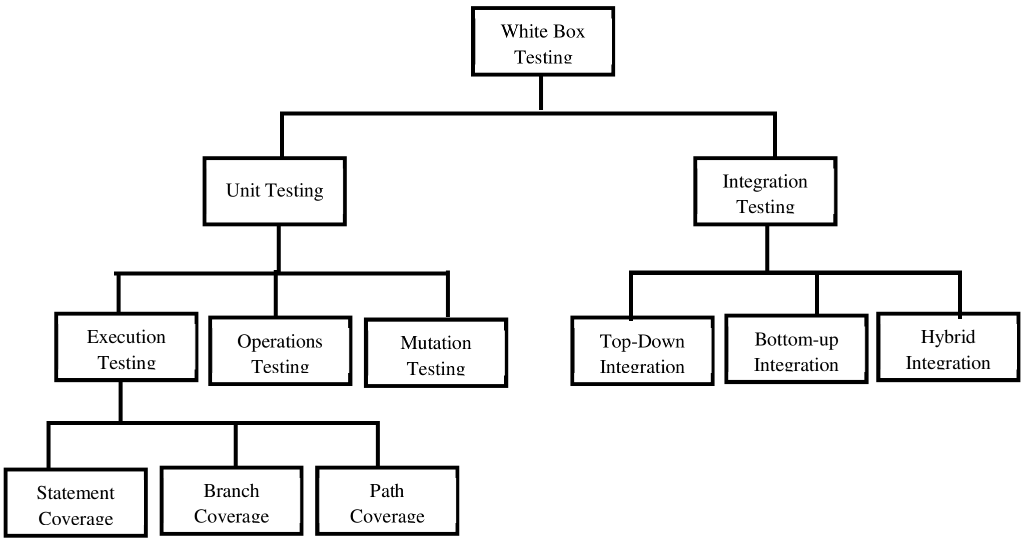
It's the counterpart of Black box testing.

In simple words – In **Black Box Testing**, we test the software from a user's point of view, but in White box, we see and test the actual code. In Black box, we do testing without seeing the internal system code, but in White box we do see and test the internal code.

White box testing technique is used by both developers as well as testers. It helps them understand which line of code is actually executed and which is not. This may indicate that there is either missing logic or a typo, which eventually can lead into some negative consequences.

Types of white box testing:

There are different types and different methods for each white box testing type. See below image



Types of white box testing

Unit Testing: Unit testing is testing of individual software modules.
Unit testing is done by the original developer. Unit Testing is done at the lowest level. It tests the basic unit of software, which is the smallest testable piece of software, and is often called "unit", "module", or "component" interchangeably.

Integration Testing: Testing during the integration of the software modules. Integration Testing is performed when two or more tested units are combined into a larger structure. The test is often done on both the interfaces between the components and the larger structure being constructed, if its quality property cannot be assessed from its components.
Performed by programmers or testing group

Objectives

- Detect interface errors
- Assure the functionality of the combined units

Integration testing involves top-down integration, bottom-up integration and hybrid integration.

Top-down integration: Develop the skeleton of the system and populate it with components.
Bottom-up integration: Integrate infrastructure components then add functional components.

Hybrid integration: This is a combination of top-down integration and bottom-up integration.

❖ SYSTEM TESTING:

Testing the software in an environment that matches the operational environment.

In system testing, the functionality, performance, reliability, and security of the entire system is checked.

System Testing tends to affirm the end-to-end quality of the entire system. System test is often based on the functional/requirement specification of the system. Non-functional requirements/quality/attributes, such as reliability, security, portability and maintainability, are also checked.
The system testing may require involvement of other systems but this should be minimized as much as possible to reduce the risk of externally-induced problems

Objectives:
- Find errors in the overall system behaviour
- Establish confidence in system functionality
- Validate non-functional system requirements

Some examples of system testing include smoke test. The first system test is often a smoke test.

This is an informal quick-and-dirty run through of the application's major functions without bothering with details.

Smoke Testing: A quick-and-dirty test that the major functions of a piece of software work without bothering with finer details. Originated in the hardware testing practice of turning on a new piece of hardware for the first time and considering it a success if it does not catch on fire.

❖ ACCEPTANCE TESTING/USER ACCEPTANCE TESTING

Testing to satisfy the purchaser - Testing to verify a product meets customer specified requirements. Tests designated by the customer to determine acceptability of the system. i.e testing done to affirm the acceptability of the system. Acceptance Testing is done when the completed system is handed over from the developers to the customers or users. A customer usually does this type of testing on the software product. The purpose of acceptance testing is rather to give confidence that the system is working than to find errors.

Operating the system in the user environment with standard user input scenario. This type of test is usually performed by the end users.

Objectives:

- Evaluate whether the system meets the customer criteria
- Determine whether the customer will accept the system

❖ PERFORMANCE TESTING

Performance testing can be applied to understand your application or web site's scalability, or to benchmark the performance in an environment of third party products such as servers and middleware for potential purchase. This sort of testing is particularly useful to identify performance bottlenecks in high use applications. Performance testing generally involves an automated test suite as this allows easy simulation of a variety of normal, peak, and exceptional load conditions. Some examples of performance testing include: Load Testing, Stress Testing, Soak Testing, Spike Testing, Configuration Testing and Isolation Testing.

(a) Load Testing: A load test is usually conducted to understand the behaviour of the system under a specific **expected load**. This load can be the **expected concurrent number of users** on the application performing a specific number of transactions within the set duration. This test will give out the response times of all the important business critical transactions.

Load testing is meant to test the system by constantly and steadily increasing the load on the system till the time it reaches the threshold limit.

(b) Stress Testing: This kind of test is done to determine the system's robustness in terms of **extreme load** and helps application administrators to determine if the system will perform sufficiently if the current load goes **well above the expected maximum. This helps the application administrator to understand the upper limits** of capacity within the system.

Under stress testing, various activities to overload the existing resources with excess jobs are carried out in an attempt to break the system down.

N/B:

- If you are testing normal expected load (for instance, you know that the system will be used by up to 100 users at a time), this is *load testing*. But when you want to determine how the system behaves under extreme load (for instance, you know that the system will be used by up to 100 users at a time, and you decided to add 10 more users), and when it breaks, this is *stress testing*.
- As an example, a word processor like Writer1.1.0 by OpenOffice.org is utilized in development of letters, presentations, spread sheets etc… Purpose of our stress testing is to load it with the excess of characters.

  To do this, we will repeatedly paste a line of data, till it reaches its threshold limit of handling large volume of text. As soon as the character size reaches 65,535 characters, it would simply refuse to accept more data. The result of stress testing on Writer 1.1.0 produces the result that, it does not crash under the stress and that it handle the situation gracefully, which make sure that application is working correctly even under rigorous stress conditions.

Basically the same thing but load is under normal load and stress is more than normal load

Performance testing can serve different purposes:

- It can demonstrate that the system meets performance criteria.
- It can compare two systems to find which performs better.
- It can measure which parts of the system or workload cause the system to perform badly.

Many performance tests are undertaken without setting sufficiently realistic, goal-oriented performance goals. The first question from a business perspective should always be, "why are we performance-testing?". These considerations are part of the business case of the testing. Performance goals will differ depending on the system's technology and purpose, but should always include some of the following:

❖ **REGRESSION TESTING**

Regression testing is a type of software testing that is done each time the system is changed. Saving tests from the previous version to ensure that the new version retains the previous capabilities. A regression test allows a consistent, repeatable validation of each new release of a product or Web site. Such testing ensures reported product defects have been corrected for each new release and that no new quality problems were introduced in the maintenance process. Though regression testing can be performed manually an automated test suite is often used to reduce the time and resources needed to perform the required testing. This type of testing is used during incremental delivery of the software product.

Regression testing is usually performed by the system itself or by a regression test group

Goal:

- To catch any new bugs introduced by change. i.e Assuring that changes to the system have not introduced new errors
- Find and eliminate newly introduced defects.

Other types of testing include:

(a) **Alpha Testing**: Testing by the customer at the developer's site.

(b) **Beta Testing**: Testing by the customer at the customer's site.

(c) COMPATIBILITY TESTING
Testing to ensure compatibility of an application or Web site with different browsers, OSs, and hardware platforms. Compatibility testing can be performed manually or can be driven by an automated functional or regression test suite.

(d) CONFORMANCE TESTING
Verifying implementation conformance to industry standards. Producing tests for the behavior of an implementation to be sure it provides the portability, interoperability, and/or compatibility a standard defines.

(e) FUNCTIONAL TESTING
Validating an application or Web site conforms to its specifications and correctly performs all its required functions. This entails a series of tests which perform a feature by feature validation of behavior, using a wide range of normal and erroneous input data. This can involve testing of the product's user interface, APIs, database management, security, installation, networking, etcF testing can be performed on an automated or manual basis using black box or white box methodologies.

Automated Software Testing Tools

Currently, there are 3,497 software tools in 51 categories. Some of them include:

- Load Runner
- Win Runner
- Rational Robot
- Visual studio Test Professional
- Network Tools
- Mobile Testing Tools
- Eclipse Plug-in Tools
- Debuggers
- Browsers Based Tools
- Cloud Based Testing Tools
- Benchmarking Tools
- Data Comparison Tools
- Link Checkers
- Performance Testing Tools
- Web Traffic Analysis Tools
- Web Hosting Tools

Assignment Three:
Differentiate between black-box Testing and White-box Testing.

LECTURE 10: DESIGN AND IMPLEMENTATION IN THE REAL WORLD

Case study 1
Case Study 2

LECTURE 11: GROUP PROJECT PRESENTATION

For the course project, students will work in small groups to design and implement any chosen professional project of their choice a

PROFESSIONAL PROJECT

- ❖ Using any project of your choice and a chosen programming language (Java, C++, VB.Net, PHP, etc) develop a software product for a nominated client.
- ❖ Each group is coordinated by a team leader who acts a project manager. Teams are required to produce a feasibility report , including Software Requirement Specification Document and equally depict the Logical design that are in accordance with currently accepted Software Engineering Principles and Concepts in order to meet the demands of the industries. These requirements must involve analysis of the project in details (Functional and Non-Functional Software Requirements)
- ❖ A requirement specification document must be produce in accordance to the functional requirements of the chosen project.
- ❖ The conceptual view of the system must be depicted using either Win Design or AlgoUML.

All Test Cases must be indicated in the logical design.

- ❖ Testing should can be conducted using automated testing tool depending on the programming language used.

Note:

- **All the projects must conform to industrial standards.**
- **Project Choice Form must be filled and return before 31th March, 2017**

INF306: Software Engineering

Academic Year 2016-2017

PROJECT CHOICE FORM

Course Code/Course Title: ***INF306: Software Engineering***	
Group Number:1	
Project Topic:	
Group Leader:	
Group Members:	
Matriculation	**Name**
(1)	
(2)	
(3)	
(4)	
(5)	

Course Code/Course Title: ***INF306: Software Engineering***	
Group Number:2	
Project Topic:	
Group Leader:	
Group Members:	
Matriculation	**Name**
(1)	
(2)	
(3)	
(4)	
(5)	