PROPOSED COURSEWARE FOR INSTRUCTIONAL DELIVERY

AL-HIKMAH UNIVERSITY, ILORIN, NIGERIA

www.alhikmah.edu.ng

COURSE TITLE: Introduction to Software Engineering

COURSE CODE: SEN 201 COURSE CREDIT: 3 COURSE STATUS: C

LECTURER'S NAME: Rahman Diekola AZEEZ

DEPARTMENT: Department of Computer Science, Faculty of Natural and Applied Sciences, Al-Hikmah University, Ilorin, Nigeria

OFFICE ROOM NUMBER: 126 FLOOR: Two

OFFICE ADDRESS: Department of Computer Science, Faculty of Natural and Applied Sciences, Adeta Campus

OFFICIAL EMAIL: rdazeez@alhikmah.edu.ng **ALTERNATIVE EMAIL:** abdulkola025@gmail.com

CONTACT MOBILE NUMBER(S): 08098488156

CONSULTATION HOURS: 10am-1pm

CONSULTATION DAYS: Mondays and Tuesdays

COURSE DETAILS: This course aims to provide a balanced approach between theoretical concepts and practical application through a variety of teaching methods and assessments. Students will engage in regular discussions, individual and group projects, as well as practical exercises to reinforce their learning.

COURSE CONTENT: Software Engineering Concepts and Principles; Design, Development and Testing of Software Systems, Introduction to Software Life Cycle. Requirements, Design and Testing. Review of Principles of Object Orientation. Object Oriented Analysis using UML. Frameworks and APIs. Introduction to the Client-Server Architecture, Analysis, Design and Programming of a team-project including user interface considerations. Use VB or Java or any HLL. (LH45:PH45)C

COURSE DESCRIPTION: This course provides an introduction to the fundamental principles and practices of software engineering. It covers the basics of software development methodologies, project management, software requirements, design, implementation, testing, and maintenance.

COURSE OBJECTIVES: At the end of this course, students are expected to:

By the end of this course, students will be able to:

1. Understand the fundamental concepts and principles of software engineering.
2. Apply different software development methodologies.
3. Analyze software requirements and design solutions accordingly.
4. Implement software solutions using appropriate programming languages and tools.
5. Conduct software testing and debugging techniques.
6. Comprehend the importance of software maintenance and version control.

COURSE REQUIREMENTS:

- ✓ **Basic knowledge of programming**
- ✓ **Access to relevant software development tools and environments**
- ✓ **Regular attendance in lectures and active participation in discussions**
- ✓ **Completion of assignments and projects on time**

Students are expected to be actively involved in classroom activities through participation, submission of assignments, asking and answering question in the lecture room, etc. Class attendance is mandatory and it will be taken at each lecture. Any student who does not meet 75% attendance would be disqualified from writing examination. Each student would be subjected to class work and practice. In addition, students will be subjected to the end of course examination using Computer Based Test of 100 multiple choice items where students is more than hundred.

GRADING METHOD: The grading system for this course would be:
Continuous Assessment

Attendance	**-----**	**10marks**
Assignment	**-----**	**10marks**
Test	**-----**	**20marks**
Examination	**-----**	**60marks**
Total	**-----**	**100marks**

INSTRUCTIONAL STRATEGY: combination of lecture and students' knowledge acquisition through class discussion and project presentations.

Learning Section 1: Introduction to Software Engineering
Learning Objective(s): At the end of this class, students should be able to identify the major terminologies in the definition of software engineering and evolution of software engineering.

Topic Outline: Overview of software engineering principles and practices, Historical perspective and evolution of software engineering, Importance of software engineering in modern technology - – Reference Text Books: Foundation of Software Engineering by Ashfaque Ahmed & Bhanu Prasad and Software Engineering by Ian Sommerville

Lecture Note
Definitions of Software, Engineering, System Engineering and Computer Science. The relationship between Computer Science and System, and Software Engineering is discussed. The human evolution of software engineering in history is also discussed:

*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.*

*So, we can define **software engineering** as an engineering branch associated with the development of software product using well-defined scientific principles, methods and procedures. The outcome of software engineering is an efficient and reliable software product.*

A brief, evolutional history of software engineering is discussed;

Human civilization has evolved from cave dwelling to modern-day city living. During the course of evolution, humans learned trades, improved their skills, and refined their living standards. A key ingredient for this evolution is the understanding of science and applying the scientific findings to improve the living standards of humans. On their way to early evolution, humans learned how to farm, use wheels, make tools, and so on, which permanently changed the way early humans lived. Since, new discoveries in science continue to change the lives of humans. Hundreds of years ago, during the Industrial Revolution, engineering started to shape human lives. Engineering helped create large man-made structures using applied science. Humans were able to build large bridges, factories, buildings, and so on, which paved the way for making modern-day infrastructures. Sometime around the 1940s, a new revolution started. This revolution was the introduction of computers. Computers are able to perform large and complex calculations quickly. With the increase in knowledge, humans are able to use computers to do many things in areas such as telecommunications, business, entertainment, technology, medicine, education, and research. Today, we use devices such as mobile phones, the Internet, and automobiles. Most of these are driven or controlled by computers.

Study Questions:
1. Explain the two (2) principles of Software Engineering.
2. Describe the evolution of software engineering.
3. Discuss the need for software engineering.

Reference Materials:

i. Reading List

ii. Web Directory

iii. YouTube Directory

Learning Section 2: Importance of software engineering in modern technology

Learning Objective(s): At the end of this lecture, student is expected to acquire the knowledge for the need for software engineering and need for quality software development process.

Topic Outline: Explanation and identification of the fundamental needs or importance of software engineering

Lecture Note

The need of software engineering arises because of higher rate of change in user requirements and environment on which the software is working.

- **Large software** - It is easier to build a wall than [to a] house or building, likewise, as the size of software become large, engineering has, to step in, to give it a scientific process.
- **Scalability** - If the software process were not based on scientific and engineering concepts, it would be easier to re-create new software than to scale an existing one.
- **Cost** - As hardware industry has shown its skills and huge manufacturing has lower down the price of computer and electronic hardware. But the cost of software remains high if proper process is not adapted.
- **Dynamic Nature** - The always growing and adapting nature of software hugely depends upon the environment in which the user works. If the nature of software is always changing, new enhancements need to be done in the existing one. This is where software engineering plays a good role.
- **Quality Management** - Better process of software development provides better and quality software product.

CHARACTERESTICS OF GOOD SOFTWARE

A software product can be judged by what it offers and how well it can be used. This software must satisfy on the following grounds:

1. Operational
2. Transitional
3. Maintenance

Well-engineered and crafted software is expected to have the following characteristics:

1. Operational

This tells us how well software works in operations. It can be measured on:

- ✓ Budget
- ✓ Usability
- ✓ Efficiency
- ✓ Correctness
- ✓ Functionality
- ✓ Dependability
- ✓ Security
- ✓ Safety

2. Transitional

This aspect is important when the software is moved from one platform to another:

- ✓ Portability
- ✓ Interoperability
- ✓ Reusability
- ✓ Adaptability

3. Maintenance

This aspect briefs about how well a software has the capabilities to maintain itself in the everchanging environment:

- ✓ Modularity
- ✓ Maintainability
- ✓ Flexibility
- ✓ Scalability

In short, Software engineering is a branch of computer science, which uses well-defined engineering concepts required to produce efficient, durable, scalable, in-budget and on-time software products.

Study Questions

1. Discuss the importance of Software Engineering
2. Explain Quality Management
3. Discuss the distinction between characteristics of a good software and importance of software engineering

Reference Materials:

i. **Reading List**

ii. **Web Materials**

iii. **YouTube Directory**

Learning Section 3: Software Development Life Cycle (SDLC)

Learning Objective(s): at the end of this lecture, students should be able to explain at least three (3) SDLC models or methods in software development

Topic Outline: SDLC definition, the reason for SDLC models, Software crisis, Understanding the principle of Code-and-Fix

Learning Note

A software life cycle model (also called process model) is a descriptive and diagrammatic representation of the software life cycle.

A life cycle model represents all the activities required to make a software product transit through its life cycle phases. It also captures the order in which these activities are to be undertaken. In other words, a life cycle model maps the different activities performed on a software product from its inception to retirement. Different life cycle models may map the basic development activities to phases in different ways. Thus, no matter which life cycle model is followed, the basic activities are included in all life cycle models though the activities may be carried out in different orders in different life cycle models. During any life cycle phase, more than one activity may also be carried out.

THE NEED FOR A SOFTWARE LIFE CYCLE MODEL

The development team must identify a suitable life cycle model for the particular project and then adhere to it. Without using of a particular life cycle model the development of a software product would not be in a systematic and disciplined manner. When a software product is being developed by a team there must be a clear understanding among team members about when and what to do. Otherwise it would lead to chaos and project failure. This problem can be illustrated by using an example. Suppose a software development problem is divided into several parts and the parts are assigned to the team members. From then on, suppose the team members are allowed the freedom to develop the parts assigned to them in whatever way they like. It is possible that one member might start writing the code for his part, another might

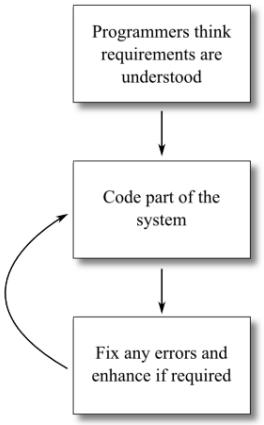
decide to prepare the test documents first, and some other engineer might begin with the design phase of the parts assigned to him. This would be one of the perfect recipes for project failure. A software life cycle model defines entry and exit criteria for every phase. A phase can start only if its phase-entry criteria have been satisfied. So without software life cycle model the entry and exit criteria for a phase cannot be recognized. Without software life cycle models it becomes difficult for software project managers to monitor the progress of the project.

THE SOFTWARE CRISIS

There were many difficulties in the development of large software systems during the 1960s and 1970s. The term **"software crisis"** dates from that time. The problems stemmed from an inability to apply the techniques used to build small software systems to the development of larger and more complex systems. The typical way to develop small systems can be described as **"code-and-fix".**

UNDERSTANDING CODE-AND-FIX SOFTWARE DEVELOPMENT APPROACH

It is important to note that the "code-and-fix" approach to software development is not a proper life cycle. Code-and-fix development occurs when software engineers come together with a vague set of requirements and start producing software, fixing it, and changing it until the correct product appears.



It is the simplest way to produce software and is invariably how every programmer learns to program. But for anything other than small software projects, code-and-fix is a disaster for a number of reasons:

i. There is no way to estimate time-scales or budgets.
ii. There is no assessment of possible risks and design flaws: you may come close to a finished product only to find an insurmountable technical problem which sets the whole project back.

Code-and-fix approach in the context of life cycle models is a base-line model which we should avoid. From a software engineer's point of view, code-and-fix is a worst case scenario.

Many project failures resulted from the inability to scale the techniques employed when developing small software systems to handle larger, more complex systems. This failure leads to:

a. never completed systems
b. missed deadlines
c. exceeded budgets
d. a system that does not do all that is required of it
e. a system that works but is difficult to use
f. a system difficult to modify to meet changes in organisational needs and practices
g. a loss of trust from users, who may experience many problems with using the software.

These problems were largely due to the lack of any framework for the planning and organisation of software development projects. Although some software projects were

organised, and these were often the more successful ones, it was the luck of the draw whether a project manager had good intuitions for software development, and whether or not problems arose due to misunderstandings between the customers and the developers of the system. Likewise, there were no clear methods to monitor whether a system was soon to go over budget or miss deadlines.
From some of these problems we can see that at some stage the system developers attempted (though not always successfully) to understand the requirements for the new system. Therefore, the need for process with specified requirements for the new system.

Study Questions

1. Briefly explain SDLC
2. Explain the four fundamentals of good software
3. Distinguish between Plan-Driven and Agile methodologies

Reference Materials:

i. **Reading List**

ii. **Web Materials**

iii. **YouTube Directory**

Learning Section 4: Software Requirements Engineering

Learning Objective(s): at the end this lecture, student should be able to acquire the skills of gathering and analyzing software requirements. Students should be able to use few use case modeling and requirement validation techniques.

Topic Outline: Software requirements Specification

Lecture Note

Software Requirements Engineering

Requirements analysis and specification: - The aim of the requirements analysis and specification phase is to understand the exact requirements of the customer and to document them properly. This phase consists of two distinct activities, namely

i. Requirements gathering and analysis
ii. Requirements specification

The goal of the requirements gathering activity is to collect all relevant information from the customer regarding the product to be developed. This is done to clearly understand the customer requirements so that incompleteness and inconsistencies are removed.
The requirements analysis activity is begun by collecting all relevant data regarding the product to be developed from the users of the product and from the customer through interviews and discussions. For example, to perform the requirements analysis of a business accounting software required by an organization, the analyst might interview all the accountants of the organization to ascertain their requirements. The data collected from such a group of users usually contain several contradictions and ambiguities, since each user typically has only a partial and incomplete view of the system. Therefore, it is necessary to identify all ambiguities and contradictions in the requirements and resolve them through further discussions with the customer. After all ambiguities, inconsistencies, and incompleteness have been resolved and all the requirements properly understood, the requirements specification activity can start. During this activity, the user requirements are systematically organized into a Software Requirements Specification (SRS) document. The customer requirements identified during the requirements gathering and analysis activity are organized into a SRS document. The important

components of this document are functional requirements, the nonfunctional requirements, and the goals of implementation.

Design: - The goal of the design phase is to transform the requirements specified in the SRS document into a structure that is suitable for implementation in some programming language. In technical terms, during the design phase the software architecture is derived from the SRS document. Two distinctly different approaches are available: the **traditional design approach** and the **object-oriented design approach**.

- **Traditional design approach** -Traditional design consists of two different activities; first a ***structured analysis*** of the requirements specification is carried out where the detailed structure of the problem is examined. This is followed by a ***structured design*** activity. During structured design, the results of structured analysis are transformed into the software design.
- **Object-oriented design approach** -In this technique, various objects that occur in the *problem domain* and the *solution domain* are first identified, and the different relationships that exist among these objects are identified. The object structure is further refined to obtain the detailed design.

Study Questions

1. Distinguish between requirement specification and gathering analysis
2. Comprehensively discuss requirement validation
3. How is Design related to SRS?

Reference Materials:

i. **Reading List**

ii. **Web Materials**

iii. **YouTube Directory**

Learning Section 5: Software Requirement continues. Functional and Non-functional requirements

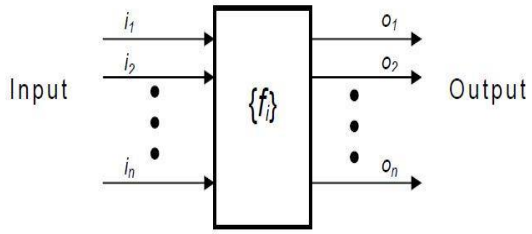
Learning Objective(s): at the end of this lecture, students should be able to distinguish between functional and non-functional requirements as well as identify the requirement problesms.

Topic Outline: Functional Requirements, Non-Functional Requirements, Problems of functional and non-functional requirements. SRS Documentation

Lecture Note

Functional requirements:-

The functional requirements part discusses the functionalities required from the system. The system is considered to perform a set of high-level functions $\{f_i\}$. The functional view of the system is shown in the figure below. Each function f_i of the system can be considered as a transformation of a set of input data (i_i) to the corresponding set of output data (o_i). The user can get some meaningful piece of work done using a high-level function.



Nonfunctional requirements:-

Nonfunctional 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.

Goals of implementation:-

The goals of implementation part documents some general suggestions regarding development. These suggestions guide trade-off among design goals. The goals of implementation section might document issues such as revisions to the system functionalities that may be required in the future, new devices to be supported in the future, reusability issues, etc. These are the items which the developers might keep in their mind during development so that the developed system may meet some aspects that are not required immediately.

Identifying functional requirements from a problem description

The high-level functional requirements often need to be identified either from an informal problem description document or from a conceptual understanding of the problem. Each high-level requirement characterizes a way of system usage by some user to perform some meaningful piece of work. There can be many types of users of a system and their requirements from the system may be very different. So, it is often useful to identify the different types of users who might use the system and then try to identify the requirements from each user's perspective.

Example: - Consider the case of the library system, where –

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.

Documenting functional requirements

For documenting the functional requirements, we need to specify the set of functionalities supported by the system. A function can be specified by identifying the state at which the data is to be input to the system, its input data domain, the output data domain, and the type of processing to be carried on the input data to obtain the output data. Let us first try to document the withdraw-cash function of an ATM (Automated Teller Machine) system. The withdraw-cash is a high-level requirement. It has several sub-requirements corresponding to the different user interactions. These different interaction sequences capture the different scenarios.

Example: - Withdraw Cash from ATM

R1: withdraw cash

Description: 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.

R1.1 select withdraw amount option

Input: "withdraw amount" option

Output: user prompted to enter the account type

R1.2: select account type
Input: user option
Output: prompt to enter amount
R1.3: get required amount
Input: amount to be withdrawn in integer values greater than 100 and less than 10,000 in multiples of 100.
Output: The requested cash and printed transaction statement.
Processing: the amount is debited from the user's account if sufficient balance is available, otherwise an error message displayed

Properties of a good SRS document

The important properties of a good SRS document are the following:

- **Concise:** The SRS document should be concise and at the same time unambiguous, consistent, and complete. Verbose and irrelevant descriptions reduce readability and also increase error possibilities.
- **Structured:** It should be well-structured. A well-structured document is easy to understand and modify. In practice, the SRS document undergoes several revisions to cope up with the customer requirements. Often, the customer requirements evolve over a period of time. Therefore, in order to make the modifications to the SRS document easy, it is important to make the document well-structured.
- **Black-box view:** It should only specify what the system should do and refrain from stating how to do these. This means that the SRS document should specify the external behavior of the system and not discuss the implementation issues. The SRS document should view the system to be developed as black box, and should specify the externally visible behavior of the system. For this reason, the SRS document is also called the black-box specification of a system.
- **Conceptual integrity:** It should show conceptual integrity so that the reader can easily understand it.
- **Response to undesired events**: It should characterize acceptable responses to undesired events. These are called system response to exceptional conditions.
- **Verifiable**: All requirements of the system as documented in the SRS document should be verifiable. This means that it should be possible to determine whether or not requirements have been met in an implementation.

Problems without a SRS document

The important problems that an organization would face if it does not develop a SRS document are as follows:

1. Without developing the SRS document, the system would not be implemented according to customer needs.
2. Software developers would not know whether what they are developing is what exactly required by the customer.
3. Without SRS document, it will be very much difficult for the maintenance engineers to understand the functionality of the system.
4. It will be very much difficult for user document writers to write the users' manuals properly without understanding the SRS document.

Problems with an unstructured specification

1. It would be very much difficult to understand that document.
2. It would be very much difficult to modify that document.
3. Conceptual integrity in that document would not be shown.
4. The SRS document might be unambiguous and inconsistent.

Study Questions

1. Distinguish between Functional and Non-Functional requirements
2. Explain the problems related to non-functional requirements.
3. Discuss SRS Documentation.

Reference Materials:

i. **Reading List**

ii. **Web Materials**

iii. **YouTube Directory**

Learning Section 6: Software Design Principles

Learning Objective(s): at the end of this lecture, students are expected to explain the principles of software design, understand UML and distinguish between structural and behavioral design concepts

Topic Outline: Software Design Principles

Lecture Note

SOFTWARE DESIGN

Software design is a process to transform user requirements into some suitable form, which helps the programmer in software coding and implementation.

For assessing user requirements, an SRS (Software Requirement Specification) document is created whereas for coding and implementation, there is a need of more specific and detailed requirements in software terms. The output of this process can directly be used into implementation in programming languages.

Software design is the first step in SDLC (Software Design Life Cycle), which moves the concentration from problem domain to solution domain. It tries to specify how to fulfill the requirements mentioned in SRS.

Software Requirement

The software requirements are description of features and functionalities of the target system. Requirements convey the expectations of users from the software product. The requirements can be obvious or hidden, known or unknown, expected or unexpected from client's point of view.

Requirement Engineering

The process to gather the software requirements from client, analyze, and document them is known as requirement engineering. The goal of requirement engineering is to develop and maintain sophisticated and descriptive 'System Requirements Specification' document.

Requirement Engineering Process

It is a four step process, which includes

i. Feasibility Study
ii. Requirement Gathering
iii. Software Requirement Specification
iv. Software Requirement Validation

Feasibility study

When the client approaches the organization for getting the desired product developed, it comes up with a rough idea about what all functions the software must perform and which all features are expected from the software.

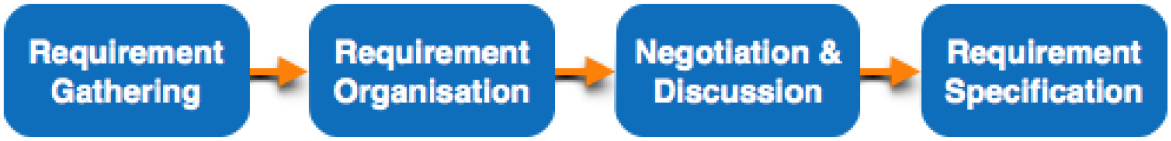
Referencing to this information, the analysts do a detailed study about whether the desired system and its functionality are feasible to develop.
This feasibility study is focused towards goal of the organization. This study analyzes whether the software product can be practically materialized in terms of implementation, contribution of project to organization, cost constraints, and as per values and objectives of the organization. It explores technical aspects of the project and product such as usability, maintainability, productivity, and integration ability.
The output of this phase should be a feasibility study report that should contain adequate comments and recommendations for management about whether or not the project should be undertaken.
Requirement Gathering
If the feasibility report is positive towards undertaking the project, next phase starts with gathering requirements from the user. Analysts and engineers communicate with the client and end-users to know their ideas on what the software should provide and which features they want the software to include.
Software Requirement Specification (SRS)
SRS is a document created by system analyst after the requirements are collected from various stakeholders.
SRS defines how the intended software will interact with hardware, external interfaces, speed of operation, response time of system, portability of software across various platforms, maintainability, speed of recovery after crashing, Security, Quality, Limitations etc.
The requirements received from client are written in natural language. It is the responsibility of the system analyst to document the requirements in technical language so that they can be comprehended and used by the software development team.
SRS should come up with the following features:

- User Requirements are expressed in natural language.
- Technical requirements are expressed in structured language, which is used inside the organization.
- Design description should be written in Pseudo code.
- Format of Forms and GUI screen prints.
- Conditional and mathematical notations for DFDs etc.

Software Requirement Validation
After requirement specifications are developed, the requirements mentioned in this document are validated. User might ask for illegal, impractical solution or experts may interpret the requirements inaccurately. This results in huge increase in cost if not nipped in the bud. Requirements can be checked against following conditions.

- ✓ If they can be practically implemented
- ✓ If they are valid and as per functionality and domain of software
- ✓ If there are any ambiguities
- ✓ If they are complete
- ✓ If they can be demonstrated

Requirement Elicitation Process
Requirement elicitation process can be depicted using the folloiwng diagram:



- ✓ Requirements gathering - The developers discuss with the client and end users and know their expectations from the software.

- ✓ Organizing Requirements - The developers prioritize and arrange the requirements in order of importance, urgency and convenience.
- ✓ Negotiation & discussion - If requirements are ambiguous or there are some conflicts in requirements of various stakeholders, it is then negotiated and discussed with the stakeholders. Requirements may then be prioritized and reasonably compromised. The requirements come from various stakeholders. To remove the ambiguity and conflicts, they are discussed for clarity and correctness. Unrealistic requirements are compromised reasonably.
- ✓ Documentation - All formal and informal, functional and non-functional requirements are documented and made available for next phase processing.

Requirement Elicitation Techniques

Requirements Elicitation is the process to find out the requirements for an intended software system by communicating with client, end users, system users, and others who have a stake in the software system development.

There are various ways to discover requirements. Some of them are explained below:

Interviews

Interviews are strong medium to collect requirements. Organization may conduct several types of interviews such as:

- ➢ Structured (closed) interviews, where every single information to gather is decided in advance, they follow pattern and matter of discussion firmly.
- ➢ Non-structured (open) interviews, where information to gather is not decided in advance, more flexible and less biased.
- ➢ Oral interviews
- ➢ Written interviews
- ➢ One-to-one interviews which are held between two persons across the table.
- ➢ Group interviews which are held between groups of participants. They help to uncover any missing requirement as numerous people are involved.

Surveys

Organization may conduct surveys among various stakeholders by querying about their expectation and requirements from the upcoming system.

Questionnaires

A document with pre-defined set of objective questions and respective options is handed over to all stakeholders to answer, which are collected and compiled. A shortcoming of this technique is, if an option for some issue is not mentioned in the questionnaire, the issue might be left unattended.

Task analysis

Team of engineers and developers may analyze the operation for which the new system is required. If the client already has some software to perform certain operation, it is studied and requirements of proposed system are collected.

Domain Analysis

Every software falls into some domain category. The expert people in the domain can be a great help to analyze general and specific requirements.

Brainstorming

An informal debate is held among various stakeholders and all their inputs are recorded for further requirements analysis.

Prototyping

Prototyping is building user interface without adding detail functionality for user to interpret the features of intended software product. It helps giving better idea of requirements. If there is no software installed at client's end for developer's reference and the client is not aware of its own requirements, the developer creates a prototype based on initially mentioned

requirements. The prototype is shown to the client and the feedback is noted. The client feedback serves as an input for requirement gathering.

Observation

Team of experts visit the client's organization or workplace. They observe the actual working of the existing installed systems. They observe the workflow at the client's end and how execution problems are dealt. The team itself draws some conclusions which aid to form requirements expected from the software.

Software Requirements Characteristics

Gathering software requirements is the foundation of the entire software development project. Hence they must be clear, correct, and well-defined.

A complete Software Requirement Specifications must be:

- ✓ Clear
- ✓ Correct
- ✓ Consistent
- ✓ Coherent
- ✓ Comprehensible
- ✓ Modifiable
- ✓ Verifiable
- ✓ Prioritized
- ✓ Unambiguous
- ✓ Traceable
- ✓ Credible source

Software Design Levels

Software design yields three levels of results:

- Architectural Design - The architectural design is the highest abstract version of the system. It identifies the software as a system with many components interacting with each other. At this level, the designers get the idea of proposed solution domain.
- High-level Design- The high-level design breaks the 'single entity-multiple component' concept of architectural design into less-abstracted view of sub-systems and modules and depicts their interaction with each other. High-level design focuses on how the system along with all of its components can be implemented in forms of modules. It recognizes modular structure of each sub-system and their relation and interaction among each other.
- Detailed Design- Detailed design deals with the implementation part of what is seen as a system and its sub-systems in the previous two designs. It is more detailed towards modules and their implementations. It defines logical structure of each module and their interfaces to communicate with other modules.

SOFTWARE DESIGN STRATEGIES [REQUIREMENTS]

Software design is a process to conceptualize the software requirements into software implementation. Software design takes the user requirements as challenges and tries to find optimum solution. While the software is being conceptualized, a plan is chalked out to find the best possible design for implementing the intended solution.

There are multiple variants of software design. Let us study them briefly:

1. Structured Design

Structured design is a conceptualization of problem into several well-organized elements of solution. It is basically concerned with the solution design. Benefit of structured design is, it gives better understanding of how the problem is being solved. Structured design also makes it simpler for designer to concentrate on the problem more accurately.

Structured design is mostly based on 'divide and conquer' strategy where a problem is broken into several small problems and each small problem is individually solved until the whole problem is solved.

The small pieces of problem are solved by means of solution modules. Structured design emphasis that these modules be well organized in order to achieve precise solution.
These modules are arranged in hierarchy. They communicate with each other. A good structured design always follows some rules for communication among multiple modules, namely –
Cohesion - grouping of all functionally related elements.
Coupling - communication between different modules.
A good structured design has high cohesion and low coupling arrangements.

2. Function Oriented Design

In function-oriented design, the system is comprised of many smaller sub-systems known as functions. These functions are capable of performing significant task in the system. The system is considered as top view of all functions.
Function oriented design inherits some properties of structured design where divide and conquer methodology is used.
This design mechanism divides the whole system into smaller functions, which provides means of abstraction by concealing the information and their operation. These functional modules can share information among themselves by means of information passing and using information available globally.
Another characteristic of functions is that when a program calls a function, the function changes the state of the program, which sometimes is not acceptable by other modules. Function oriented design works well where the system state does not matter and program/functions work on input rather than on a state.
Design Process (Functional)

i. The whole system is seen as how data flows in the system by means of data flow diagram.
ii. DFD depicts how functions change the data and state of entire system.
iii. The entire system is logically broken down into smaller units known as functions on the basis of their operation in the system.
iv. Each function is then described at large.

3. Non-Functional

Requirements, which are not related to functional aspect of software, fall into this category. They are implicit or expected characteristics of software, which users make assumption of.
Non-functional requirements include:

- ✓ Security
- ✓ Logging
- ✓ Storage
- ✓ Configuration
- ✓ Performance
- ✓ Cost
- ✓ Interoperability
- ✓ Flexibility
- ✓ Disaster recovery
- ✓ Accessibility

Study Questions

1. Discuss UML
2. Explain Software Design
3. Explain the concept of structural design

Reference Materials:

i. Reading List

ii. Web Materials

iii. YouTube Directory

Learning Section 7: Software Implementation Testing

Learning Objective(s): At the end of the lecture, students should be able acquire enriched knowledge on coding best practices and standards. Testing strategies: unit testing, integration testing and system testing.

Topic Outline: Test-driven development (TDD), Unit, Integration/System Testing.

Lecture Note

What is Test Driven Development (TDD)?

In layman's terms, Test Driven Development (TDD) is a software development practice that focuses on creating unit test cases before developing the actual code. It is an iterative approach combining programming, unit test creation, and refactoring.

- The TDD approach originates from the Agile manifesto principles and Extreme programming.
- As the name suggests, the test process drives software development.
- Moreover, it's a structuring practice that enables developers and testers to obtain optimized code that proves resilient in the long term.
- In TDD, developers create small test cases for every feature based on their initial understanding. The primary intention of this technique is to modify or write new code only if the tests fail. This prevents duplication of test scripts.

Test Driven Development (TDD) Examples

1. **Calculator Function**: When building a calculator function, a TDD approach would involve writing a test case for the "add" function and then writing the code for the process to pass that test. Once the "add" function is working correctly, additional test cases would be written for other functions such as "subtract", "multiply" and "divide".
2. **User Authentication**: When building a user authentication system, a TDD approach would involve writing a test case for the user login functionality and then writing the code for the login process to pass that test. Once the login functionality works correctly, additional test cases will be written for registration, password reset, and account verification.
3. **E-commerce Website**: When building an e-commerce website, a TDD approach would involve writing test cases for various features such as product listings, shopping cart functionality, and checkout process. Tests would be written to ensure the system works correctly at each process stage, from adding items to the cart to completing the purchase.

TDD Vs. Traditional Testing

- **Approach**: TDD is an agile development methodology where tests are written before the code is developed. In contrast, traditional testing is performed after the code is written.
- **Testing Scope**: TDD focuses on testing small code units at a time, while traditional testing covers testing the system as a whole, including integration, functional, and acceptance testing.
- **Iterative**: TDD follows an iterative process, where small chunks of code are developed, tested, and refined until they pass all tests. The code is usually tested once and then refined based on the results in traditional testing.

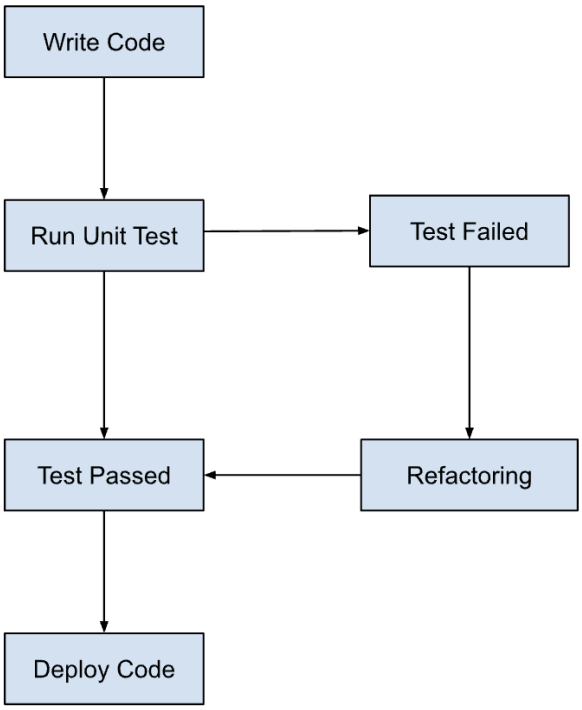
- **Debugging**: TDD aims to catch errors as early as possible in the development process, making debugging and fixing them easier. Traditional testing, on the other hand, may require more effort to debug errors that are discovered later in the development process.
- **Documentation**: TDD documentation typically focuses on the test cases and their results, while traditional testing documentation may include more detailed information about the testing process, the test environment, and the system under test.

Overall, TDD offers a more efficient and reliable approach to software development, ensuring that code is thoroughly tested before being integrated into the system. Traditional testing, however, may be more appropriate for larger and more complex projects where a more comprehensive approach to testing is required.

Three Phases of Test Driven Development

1. **Create precise tests:** Developers need to create exact unit tests to verify the functionality of specific features. They must ensure that the test compiles so that it can execute. In most cases, the test is bound to fail. This is a meaningful failure as developers create compact tests based on their assumptions of how the feature will behave.
2. **Correcting the Code:** Once a test fails, developers must make the minimal changes required to update the code to run successfully when re-executed.
3. **Refactor the Code:** Once the test runs successfully, check for redundancy or any possible code optimizations to enhance overall performance. Ensure that refactoring does not affect the external behavior of the program.

The image below represents a high-level TDD approach toward development:



Also Read: TDD vs BDD vs ATDD

How TDD fits into Agile development?

Agile development demands regular feedback to develop the expected product. In simple terms, one can also term Agile development as Feedback Driven Development.

There's a high probability that project requirements may change during the development sprint cycle. To deal with this and to build products aligned with the client's changing requirements, teams need constant feedback to avoid dishing out unusable software. TDD is built to offer such feedback early on.

TDD's test-first approach also helps mitigate critical bottlenecks that obstruct the quality and delivery of software. Based on the constant feedback, bug fixes, and the addition of new features, the system evolves to ensure that everything works as intended. TDD enhances collaboration between team members from both the development and QA teams and the client. Additionally, as the tests are created beforehand, teams don't need to spend time recreating extensive test scripts.

Benefits of Test Driven Development (TDD)

1. Fosters the creation of optimized code.
2. It helps developers better analyze and understand client requirements and request clarity when not adequately defined.
3. Adding and testing new functionalities become much easier in the latter stages of development.
4. Test coverage under TDD is much higher compared to conventional development models. The TDD focuses on creating tests for each functionality right from the beginning.
5. It enhances the productivity of the developer and leads to the development of a codebase that is flexible and easy to maintain.

Frameworks for Test Driven Development

Based on unique programming languages, multiple frameworks support test driven development. Listed below are a few popular ones.

1. **csUnit and NUnit** are open source unit testing frameworks for .NET projects.
2. **PyUnit and DocTest:** Popular Unit testing framework for Python.
3. **Junit:** Widely used unit testing tool for Java
4. **TestNG:** Another popular Java testing framework. This framework overcomes the limitations of Junit.
5. **Rspec:** A testing framework for Ruby projects

Best Practices for Test Driven Development (TDD)

Test-driven development (TDD) is a software development practice that emphasizes writing tests before writing the actual code. It follows a cyclical process of writing a failing test, writing the minimum code to make the test pass, and then refactoring the code. Here are some best practices to consider when practicing TDD:

1. **Start with a clear understanding of requirements:** Begin by understanding the requirements or specifications of the feature you are developing. This will help you write focused and relevant tests.
2. **Write atomic tests:** Each test should focus on a specific behavior or functionality. Keep your tests small and focused, addressing a single aspect of the code. This improves test readability, maintainability, and allows for easier debugging.
3. **Write the simplest test case first:** Begin by writing the simplest possible test case that will fail. This helps you focus on the immediate task and avoids overwhelming yourself with complex scenarios upfront.
4. **Write tests for edge cases:** Consider boundary conditions and edge cases when designing your tests. These are inputs or scenarios that lie at the extremes of the input domain and often reveal potential bugs or unexpected behavior.
5. **Refactor regularly:** After a test passes, take time to refactor the code and improve its design without changing its behavior. This helps maintain clean and maintainable code as the project progresses.
6. **Maintain a fast feedback loop:** Your test suite should execute quickly so that you can receive immediate feedback on the health of your code. Fast feedback allows for faster development iterations and catches issues early on.

7. **Automate your tests:** Utilize test automation frameworks and tools to automate the execution of your tests. This enables you to run tests frequently, easily integrate them into your development workflow, and ensure consistent and reliable test results.
8. **Follow the Red-Green-Refactor cycle:** Adhere to the core TDD cycle of writing a failing test (Red), implementing the minimum code to pass the test (Green), and then refactoring the code to improve its design (Refactor). Repeat this cycle for each new behavior or feature.
9. **Maintain a comprehensive test suite:** Aim to achieve a good balance between unit tests, integration tests, and acceptance tests. Each test type serves a different purpose and provides different levels of confidence in the code.
10. **Continuously run tests:** Integrate your test suite with your development environment and set up continuous integration (CI) pipelines to automatically execute tests whenever code changes are made. This ensures that tests are run consistently and helps catch regressions early.
11. **Test failures should guide development:** When a test fails, it should guide your development efforts. Analyze the failure, identify the cause, and fix the code to address the issue. Test failures are valuable feedback for improving code quality.

Delivering quality products requires debugging and optimization in the development process. When incorporated correctly, the TDD approach provides numerous benefits, particularly in bringing cost-efficiency in the long run and delivering true value to businesses.

Study Questions

1. Describe Software Implementation
2. Explain TDD
3. Distinguish between the following; unit testing, integration/system testing

Reference Materials:

i. **Reading List**

ii. **Web Materials**

iii. **YouTube Directory**

Learning Section 8: Software Maintenance and Configuration Management

Learning Objective(s): At the end of the lecture, students should be able convincingly discuss both Software Maintenance and Configuration Management.

Topic Outline: Software maintenance, Configuration management.

Lecture Note

Software maintenance is the process of changing, modifying, and updating software to keep up with customer needs. Software maintenance is done after the product has launched for several reasons including improving the software overall, correcting issues or bugs, to boost performance, and more.

Software maintenance is a natural part of SDLC (software development life cycle). Software developers don't have the luxury of launching a product and letting it run, they constantly need to be on the lookout to both correct and improve their software to remain competitive and relevant.

Using the right software maintenance techniques and strategies is a critical part of keeping any software running for a long period of time and keeping customers and users happy.
Why is software maintenance important?

Creating a new piece of software and launching it into the world is an exciting step for any company. A lot goes into creating your software and its launch including the actual building and coding, licensing models, marketing, and more. However, any great piece of software must be able to adapt to the times.

This means monitoring and maintaining properly. As technology is changing at the speed of light, software must keep up with the market changes and demands.
What are the 4 types of software maintenance?

The four different types of software maintenance are each performed for different reasons and purposes. A given piece of software may have to undergo one, two, or all types of maintenance throughout its lifespan.

The four types are:

1. Corrective Software Maintenance
2. Preventative Software Maintenance
3. Perfective Software Maintenance
4. Adaptive Software Maintenance

Corrective Software Maintenance

Corrective software maintenance is the typical, classic form of maintenance (for software and anything else for that matter). Corrective software maintenance is necessary when something goes wrong in a piece of software including faults and errors. These can have a widespread impact on the functionality of the software in general and therefore must be addressed as quickly as possible.

Many times, software vendors can address issues that require corrective maintenance due to bug reports that users send in. If a company can recognize and take care of faults before users discover them, this is an added advantage that will make your company seem more reputable and reliable (no one likes an error message after all).

Preventative Software Maintenance

Preventative software maintenance is looking into the future so that your software can keep working as desired for as long as possible.

This includes making necessary changes, upgrades, adaptations and more. Preventative software maintenance may address small issues which at the given time may lack significance but may turn into larger problems in the future. These are called latent faults which need to be detected and corrected to make sure that they won't turn into effective faults.

Perfective Software Maintenance

As with any product on the market, once the software is released to the public, new issues and ideas come to the surface. Users may see the need for new features or requirements that they

would like to see in the software to make it the best tool available for their needs. This is when perfective software maintenance comes into play.

Perfective software maintenance aims to adjust software by adding new features as necessary and removing features that are irrelevant or not effective in the given software. This process keeps software relevant as the market, and user needs, change.

Adaptive Software Maintenance

Adaptive software maintenance has to do with the changing technologies as well as policies and rules regarding your software. These include operating system changes, cloud storage, hardware, etc. When these changes are performed, your software must adapt in order to properly meet new requirements and continue to run well.

Study Questions

1. Explain Software Maintenance
2. Discuss the significance of software documentation
3. Explain two (2) phases of software maintenance

Reference Materials:

i. **Reading List**

ii. **Web Materials**

iii. **YouTube Directory**

Learning Section 9: Software Configuration

Learning Objective(s): At the end of the lecture, students should be to differentiate between Software configuration and Maintenance

Topic Outline: Software Configuration

Lecture Note

Software Configuration Management

The software configuration management process is considered by many IT personnel as the best solution for handling changes to software projects. The software configuration management process identifies the functional and physical attributes of software at critical points in time, and implements procedures to control changes to an identified attribute with the objective of maintaining software integrity and traceability throughout the software life cycle.

The software configuration management process traces changes and verifies that the software has all of the planned changes that are supposed to be included in a new release. It includes four procedures that should be defined for each software project to ensure that a reliable software configuration management process is utilized. The four procedures typically found in a reliable software configuration management system are:

1. Configuration identification

2. Configuration control

3. Configuration status documentation
4. Configuration audits

These procedures have different names in various configuration management standards but the definitions are basically the same across standards.

- ✓ *Configuration identification* is the procedure by which attributes are identified that defines all the properties of a configuration item.

A configuration item referred to as an object is a product (hardware and/or software) that supports use by an end user. These attributes are recorded in configuration documents or database tables and baselined. A baseline is an approved configuration object, such as a project plan, that has been authorized for implementation. Usually a baseline is a single work product or set of work products that can be used as a logical basis for comparison. A baseline may also be established as the basis for future activities. The configuration of a project often includes one or more baselines, the status of the configuration, and any measurement data collected. A current configuration refers to the current status, current audit, current measurements, and the latest revision of all configuration objects. Sometimes a baseline may refer to all objects associated with a specific project. This may include all revisions made to all objects, or only the latest revision of objects in a project, depending on the context in which the term *baseline* is used.

Baselining a project attribute forces formal configuration change control processes to be enacted in the event that these attributes are changed. A baseline may also be specialized as a specific type of baseline, such as

1. *Functional baseline*—initial specifications, contract specifications, regulations, design specifications, etc.
2. *Allocated baseline*—state of work products once requirements have been approved.
3. *Developmental baseline*—status of work products while in development.
4. *Product baseline*—contains the contents of the project to be released.
5. Others, based on individual business practices.

- ✓ *Configuration change control* is a set of processes and approval stages required to change a configuration object's attributes and to rebaseline them.
- ✓ *Configuration status accounting* is the ability to record and report on the configuration baselines associated with each configuration object at any point in time.

- ✓ *Configuration audits* are divided into functional and physical configuration audits. An audit occurs at the time of delivery of a project or at the time a change is made. A functional configuration audit is intended to make sure that functional and performance attributes of a configuration object are achieved. A physical configuration audit attempts to ensure that a configuration object is installed based on the requirements of its design specifications.

Study Questions

1. Explain Software Configuration
2. Discuss the significance of software configuration

Reference Materials:

- **i. Reading List**
- **ii. Web Materials**
- **iii. YouTube Directory**

Learning Section 10: Project Management in Software Engineering

Learning Objective(s): At the end of this lecture, students should be able to discuss the importance of project management in software engineering.

Topic Outline: understanding project management, Activities of project manager, project planning, Risk management

Lecture Note

What is Project?
A project is a group of tasks that need to complete to reach a clear result. A project also defines as a set of inputs and outputs which are required to achieve a goal. Projects can vary from simple to difficult and can be operated by one person or a hundred.

Projects usually described and approved by a project manager or team executive. They go beyond their expectations and objects, and it's up to the team to handle logistics and complete the project on time. For good project development, some teams split the project into specific tasks so they can manage responsibility and utilize team strengths.

What is software project management?

Software project management is an art and discipline of planning and supervising software projects. It is a sub-discipline of software project management in which software projects planned, implemented, monitored and controlled.
It is a procedure of managing, allocating and timing resources to develop computer software that fulfills requirements.

In software Project Management, the client and the developers need to know the length, period and cost of the project.
Prerequisite of software project management?

There are three needs for software project management. These are:

1. Time
2. Cost
3. Quality

It is an essential part of the software organization to deliver a quality product, keeping the cost within the client?s budget and deliver the project as per schedule. There are various factors, both external and internal, which may impact this triple factor. Any of three-factor can severely affect the other two.

Project Manager

A project manager is a character who has the overall responsibility for the planning, design, execution, monitoring, controlling and closure of a project. A project manager represents an essential role in the achievement of the projects.

A project manager is a character who is responsible for giving decisions, both large and small projects. The project manager is used to manage the risk and minimize uncertainty. Every decision the project manager makes must directly profit their project.

Role of a Project Manager:

1. Leader

A project manager must lead his team and should provide them direction to make them understand what is expected from all of them.

2. Medium:

The Project manager is a medium between his clients and his team. He must coordinate and transfer all the appropriate information from the clients to his team and report to the senior management.

3. Mentor:

He should be there to guide his team at each step and make sure that the team has an attachment. He provides a recommendation to his team and points them in the right direction.

Responsibilities of a Project Manager:

1. Managing risks and issues.
2. Create the project team and assigns tasks to several team members.
3. Activity planning and sequencing.
4. Monitoring and reporting progress.
5. Modifies the project plan to deal with the situation.

Activities

Software Project Management consists of many activities, that includes planning of the project, deciding the scope of product, estimation of cost in different terms, scheduling of tasks, etc.

The list of activities are as follows:

1. Project planning and Tracking
2. Project Resource Management
3. Scope Management
4. Estimation Management
5. Project Risk Management
6. Scheduling Management
7. Project Communication Management
8. Configuration Management

Now we will discuss all these activities -

1. Project Planning: It is a set of multiple processes, or we can say that it a task that performed before the construction of the product starts.

2. Scope Management: It describes the scope of the project. Scope management is important because it clearly defines what would do and what would not. Scope Management create the project to contain restricted and quantitative tasks, which may merely be documented and successively avoids price and time overrun.

3. Estimation management: This is not only about cost estimation because whenever we start to develop software, but we also figure out their size(line of code), efforts, time as well as cost.
If we talk about the size, then Line of code depends upon user or software requirement.
If we talk about effort, we should know about the size of the software, because based on the size we can quickly estimate how big team required to produce the software.
If we talk about time, when size and efforts are estimated, the time required to develop the software can easily determine.

And if we talk about cost, it includes all the elements such as:

- Size of software
- Quality
- Hardware
- Communication
- Training
- Additional Software and tools
- Skilled manpower

4. Scheduling Management: Scheduling Management in software refers to all the activities to complete in the specified order and within time slotted to each activity. Project managers define multiple tasks and arrange them keeping various factors in mind.
For scheduling, it is compulsory -

- Find out multiple tasks and correlate them.
- Divide time into units.
- Assign the respective number of work-units for every job.
- Calculate the total time from start to finish.
- Break down the project into modules.

5. Project Resource Management: In software Development, all the elements are referred to as resources for the project. It can be a human resource, productive tools, and libraries.
Resource management includes:

- Create a project team and assign responsibilities to every team member
- Developing a resource plan is derived from the project plan.
- Adjustment of resources.

6. Project Risk Management: Risk management consists of all the activities like identification, analyzing and preparing the plan for predictable and unpredictable risk in the project.
Several points show the risks in the project:

- The Experienced team leaves the project, and the new team joins it.
- Changes in requirement.
- Change in technologies and the environment.
- Market competition.

7. Project Communication Management: Communication is an essential factor in the success of the project. It is a bridge between client, organization, team members and as well as other stakeholders of the project such as hardware suppliers.

From the planning to closure, communication plays a vital role. In all the phases, communication must be clear and understood. Miscommunication can create a big blunder in the project.

8. Project Configuration Management: Configuration management is about to control the changes in software like requirements, design, and development of the product.
The Primary goal is to increase productivity with fewer errors.

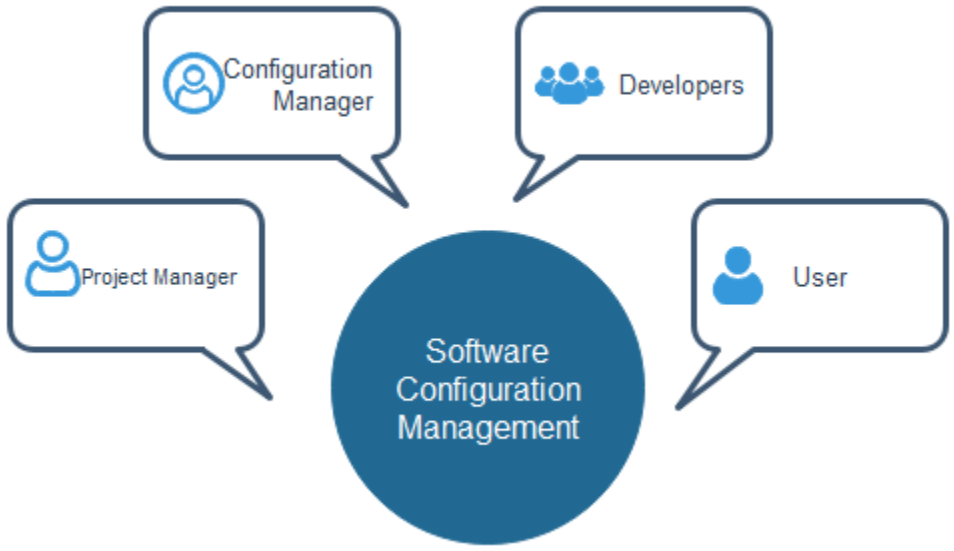
Some reasons show the need for configuration management:

- Several people work on software that is continually update.
- Help to build coordination among suppliers.
- Changes in requirement, budget, schedule need to accommodate.
- Software should run on multiple systems.

Tasks perform in Configuration management:

- Identification
- Baseline
- Change Control
- Configuration Status Accounting
- Configuration Audits and Reviews

People involved in Configuration Management:



Study Questions

1. Discuss Scrum as a Project Management methodologies
2. Explain Risk Management in software projects
3. Explain software project planning.

Reference Materials:

i. **Reading List**

ii. **Web Materials**

iii. **YouTube Directory**

Learning Section 11: Agile Development Approach

Learning Objective(s): At the end of the lecture, students should be able to explain the Agile approach to software development.

Topic Outline: Agile System approach

Lecture Note

Agile Software Development Approach

The agile approach for software development has been applied more and more extensively since the mid nineties of the 20th century. Though there are only about ten years of accumulated experience using the agile approach, it is currently conceived as one of the mainstream approaches for software development.

Agile software development refers to methods and practices that provide value quickly, efficiently, and consistently to customers as it relates to the software development lifecycle (SDLC). The ability to build and react to change is called *Agile*. It is a method for coping and, ultimately, succeeding in an uncertain and turbulent development environment. Within the Agile software development model, self-organizing and cross-functional teams work together to build and deploy solutions. Some of the popular Agile methodologies include Scrum, Kanban, and Lean.

Note:

Scrum is an agile framework used in project management and software development. It emphasizes collaboration, iterative progress, and flexibility. It organizes work into time-bound iterations called "sprints," typically lasting 1 to 4 weeks. During each sprint, a cross-functional team works on a set of prioritized tasks, with a goal of delivering a potentially shippable product increment. Regular meetings, such as daily stand-ups and sprint reviews, facilitate communication and adaptability throughout the process.

Agile splits enormous software development tasks into smaller, more manageable parts called ***iterations***. Agile differs from other software development methodologies in that it focuses mainly on the people performing the job and the cooperation, collaboration, and communication between them. Solutions emerge from collaboration amongst self-organizing cross-functional teams that use the best techniques available.

Instead of writing one extensive and comprehensive application, the Agile method involves breaking software applications into smaller, more manageable pieces. These pieces are built, tested, and deployed in iterations. Iterations are short, time-boxed periods during which you change your application's features and functionality. Most importantly, you can implement these changes quickly.

What is Agile Software Development?

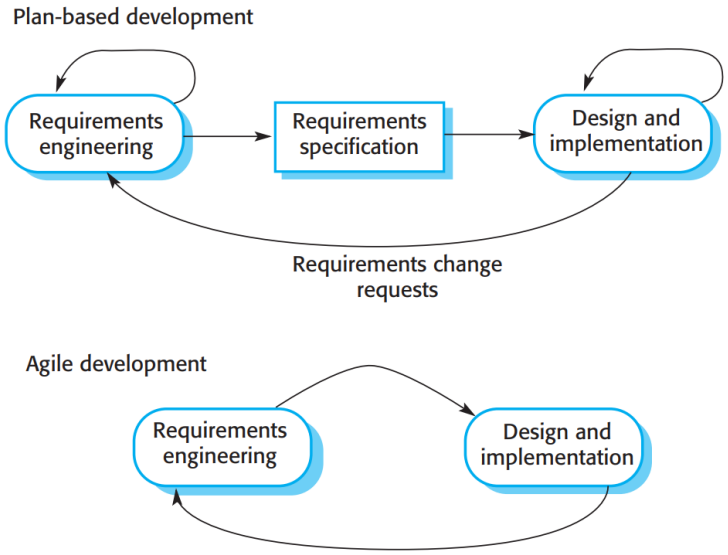
The Agile software development paradigm is a software development methodology comprising practices and approaches that thrive on iterative and incremental software development. In this development methodology, the requirements – as well as the solutions – evolve through collaboration amongst self-organizing, cross-functional programmer teams that may not or may not be collocated or remote.

The Agile approach fosters adaptability, evolutionary development, and delivery, as well as a time-bound, iterative approach and quick response to change. Agile promotes adaptive planning, evolutionary growth, early delivery, and continual improvement.

Illustration

Agile methods are incremental development methods in which the increments are small, and, typically, new releases of the system are created and made available to customers every two or three weeks. They involve customers in the development process to get rapid feedback on changing requirements. They minimize documentation by using informal communications

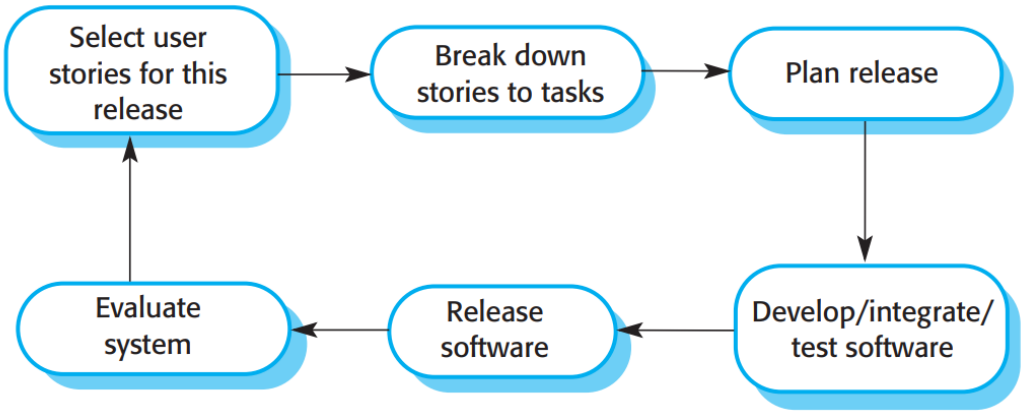
rather than formal meetings with written documents. Agile approaches to software development consider design and implementation to be the central activities in the software process. They incorporate other activities, such as requirements elicitation and testing, into design and implementation.



Agile methods have been particularly successful for two kinds of system development.

1. Product development where a software company is developing a small or medium-sized product for sale. Virtually all software products and apps are now developed using an agile approach. E.g. spotify – popular music streaming service (scrum and Kanban) and also, JIRA – issue tracking tool developed by Atlassian (project management)
2. Custom system development within an organization, where there is a clear commitment from the customer to become involved in the development process and where there are few external stakeholders and regulations that affect the software. E.g, most CRM like Sale-force customization and HRIMS

The most significant approach to changing software development culture was the development of Extreme Programming (XP). The name was coined by Kent Beck (Beck 1998) because the approach was developed by pushing recognized good practice, such as iterative development, to "extreme" levels. For example, in XP, several new versions of a system may be developed by different programmers, integrated, and tested in a day.



In XP, requirements are expressed as scenarios (called user stories), which are implemented directly as a series of tasks. Programmers work in pairs and develop tests for each task before writing the code. All tests must be successfully executed when new code is integrated into the system. There is a short time gap between releases of the system. Extreme programming was controversial as it introduced a number of agile practices that were quite different from the

development practice of that time. These practices are summarized in the table below and reflect the five (5) principles of the agile manifesto:

1. **Incremental development** is supported through small, frequent releases of the system. Requirements are based on simple customer stories or scenarios that are used as a basis for deciding what functionality should be included in a system increment.
2. **Customer involvement** is supported through the continuous engagement of the customer in the development team. The customer representative takes part in the development and is responsible for defining acceptance tests for the system.
3. **People, not process**, are supported through pair programming, collective ownership of the system code, and a sustainable development process that does not involve excessively long working hours.
4. **Change is embraced** through regular system releases to customers, test-first development, refactoring to avoid code degeneration, and continuous integration of new functionality.
5. **Maintaining simplicity** is supported by constant refactoring that improves code quality and by using simple designs that do not unnecessarily anticipate future changes to the system.

Principle or practice	Description
Collective ownership	The pairs of developers work on all areas of the system, so that no islands of expertise develop and all the developers take responsibility for all of the code. Anyone can change anything.
Continuous integration	As soon as the work on a task is complete, it is integrated into the whole system. After any such integration, all the unit tests in the system must pass.
Incremental planning	Requirements are recorded on "story cards," and the stories to be included in a release are determined by the time available and their relative priority. The developers break these stories into development "tasks." See Figures 3.5 and 3.6.
On-site customer	A representative of the end-user of the system (the Customer) should be available full time for the use of the XP team. In an extreme programming process, the customer is a member of the development team and is responsible for bringing system requirements to the team for implementation.
Pair programming	Developers work in pairs, checking each other's work and providing the support to always do a good job.
Refactoring	All developers are expected to refactor the code continuously as soon as potential code improvements are found. This keeps the code simple and maintainable.
Simple design	Enough design is carried out to meet the current requirements and no more.
Small releases	The minimal useful set of functionality that provides business value is developed first. Releases of the system are frequent and incrementally add functionality to the first release.
Sustainable pace	Large amounts of overtime are not considered acceptable, as the net effect is often to reduce code quality and medium-term productivity.
Test first development	An automated unit test framework is used to write tests for a new piece of functionality before that functionality itself is implemented.

Why Should I Use Agile?

Agile was established for the software industry to simplify and expedite the development process to detect and correct errors and difficulties swiftly. It enables teams and developers to produce a better product in less time by using short, iterative *sprints* or sessions. And, with more businesses transitioning to the digital workplace, being Agile is a perfect match for enterprises aiming to improve the way they generally manage projects and function.

The most significant benefit of the Agile development methodology is that it fosters shorter development cycles. So, you can get your products or features to market faster and start seeing returns on your investment sooner.
Agile also enables more collaboration between different team members, leading to a more cohesive team and better products. And finally, Agile allows you to get better customer feedback, which can help you make better decisions about what to build next.
Here are the benefits of the Agile methodology:

- Shorter development cycles
- Better flexibility
- Faster, incremental releases
- Adaptability
- Improved quality
- Better risk management
- Reduced costs

What is The Agile Lifecycle?
A typical Agile lifecycle consists of the following steps:

1. **Project Planning** – This helps your team understand the goals, the value to be delivered to the stakeholders, and define the project scope.
2. **Product Roadmap** – This helps to define a breakdown of all the features that are needed as part of the final deliverable.
3. **Release Planning** – This helps plan the future releases and revisit and reevaluate the release plans before a sprint starts.
4. **Sprint Planning** – This helps plan how the tasks in a sprint should be accomplished, by whom and the time it would take to complete those tasks.
5. **Daily Scrum Meetings** – These meetings are usually short and help the team know the tasks to be accomplished on a particular day, the roadblocks (if any) and assess if any changes are required.
6. **Sprint Review / Retrospective Meetings** – These meetings are usually held after every sprint to discuss what went well in the sprint and what did not or what could have been done better.

The principles outlined in the Agile Manifesto enable businesses to strive towards excellence. The manifesto promotes trust, transparency, collaboration, and consumer involvement. While software strategies may seem basic, it is always critical to work with experts that understand the significance of delivering a quality product and maintaining customer satisfaction. The Agile Manifesto has four key values:

1. Individuals and interactions over processes and tools
2. Working software over comprehensive documentation
3. Customer collaboration over contract negotiation
4. Responding to change over following a plan

What are Distributed Agile Teams?
Although communication is seamless in co-located teams, you may not have all the team members with the necessary expertise in the same location. Today's development firms may have dispersed teams spread across the world.
Distributed Agile teams can help you lower costs and accelerate time to market while opening your options to allow you to have motivated, talented people spread across the globe.

Study Questions

1. Describe the Agile Software Development Approach
2. Explain distributed agile team.
3. Discuss Xtreme Programming

Reference Materials:

i. **Reading List**

ii. **Web Materials**

iii. **YouTube Directory**

Learning Section 12: Distinguishing between different approaches in Software Engineering

Learning Objective(s): At the end of the lecture, students should be able to differentiate the different developmental approaches in software engineering.

Topic Outline: Case study: Agile Versus Plan-Driven Approach

Lecture Note
Agile versus Waterfall Methodology

Changes may be made far in advance in the Agile methodology while staying within the project budget. Even when the scope is not predefined, the Agile method works well. On the contrary, the Waterfall method works well *only* when the project scope is well-defined in advance.
Typically, features are prioritized in Agile, and concerns are addressed according to their priority. This improves financing efficiency while avoiding project failures entirely. On the other hand, the Waterfall technique never prioritizes features, which results in either complete success or failure.

The Agile approach helps modifications occur intermittently throughout a process. The Waterfall approach does not enable adjustments throughout the project process, and, if a mistake is made, the project must be restarted from the beginning. Customers may be reached throughout the project thanks to the Agile methodology best practices. Customers' availability is only required in stages in Waterfall.

Summary of Agile Software Development
Agile fosters collaboration, responsiveness to change, simplicity, and self-organization. It is a flexible, iterative, and incremental methodology that allows flexibility and adaptation to change. However, being Agile requires transparency and trust, and building this trust takes time. It would help if you had the right people who are motivated, organized, and who are willing to adhere to Agile principles. An Agile software development team that delivers value consistently can bring business agility to an enterprise.

Study Questions

1. Differentiate between Agile and Plan-driven software development approach. Use any process model to substantiate your argument.

Reference Materials:

i. **Reading List**

ii. **Web Materials**

iii. **YouTube Directory**

Learning Section 13: Revision