18CCP301 **SOFTWARE MODELS AND ENGINEERING**
4H –4C

Instruction Hours/ week: **L: 4 T: 0 P : 0** **Marks: Internal: 40** **External: 60**
Total: 100

UNIT–I

Fundamentals of Software Engineering and Process models : Definition, Software characteristics and Application. Software myths, Software engineering- A layered technology and SDLC. Software process models: Linear sequential model, prototyping model, RAD Model. Evolutionary process models: Incremental process models and Spiral model. Component based ,4GT. Maturity Models: CMM, CMMI, PCMM, PSP, TSP, Process patterns, process assessment. Unified process: SEI CMM and ISO 9001. PSP and Six Sigma. Clean room technique.

UNIT–II

Managing Software Projects & Design Engineering: The management spectrum, software quality, measurement and metrics. Software project estimation, decomposition techniques. Empirical estimation models(COCOMO), the Make & Buy Decision. System models: Context Models, Behavioral models, Data models, Object models. Design process, Design quality and design model. Fundamental issues in software design: Goodness of design, cohesions, coupling. Function-oriented design and object – oriented concepts. Architectural styles and patterns, Architectural Design: Unified Modeling Language (UML), User interface design. Risk Analysis and management.

UNIT–III

S/W Requirements, S/W Metrices & Testing Strategies: S/W Requirements : Functional and non-functional requirements, User requirements, System requirements.SRA & SRS. S/W Metrices: Process Metrices, Project Metrices & Product Metrices. Testing Strategies : A strategic approach to software testing, Testing fundamentals, Test Case Design. Types Of Testing: Black-Box Testing, White-Box Testing, Validation testing, System testing, the art of Debugging. Code walkthrough and reviews. Software Quality, Metrics for Analysis Model, Metrics for Design Model, Metrics for source code, Metrics for testing, Metrics for maintenance.

UNIT-IV

Testing Plan and Maintenance: Snooping for information, Coping with complexity through teaming, Testing plan focus areas, Testing for recoverability, Planning for troubles, Preparing for the tests: Software Reuse, Developing good test programs , Data corruption, Tools, Test Execution ,Testing with a virtual computer, Simulation and Prototypes, Managing the Test, Customer's role in testing, Software maintenance issues and techniques. Software reuse. Client-Server software development.

UNIT–V

Software Reengineering and Project Management: Software Reengineering, Reverse Engineering & Forward Engineering, Life Cycle Phases and Process artifacts, Restructuring. Model based software architectures, Software process and Iteration workflows, Major and Minor milestones, Periodic status assessments, Process Planning, Project Control and process instrumentation: Seven core metrics, management indicators, quality indicators, life-cycle expectations, CCPDS-R Case Study and Future Software Project Management Practices.

KARPAGAM ACADEMY OF HIGHER EDUCATION

(Deemed to be University Established Under Section 3 of UGC Act 1956)

Coimbatore – 641 021.

Lecture Plan

DEPARTMENT OF COMMERCE (Computer Applications)

STAFF NAME: JOTHISH.C

SUBJECT NAME:SOFTWARE MODELS AND ENGINEERING

SUB CODE: 18CCP301SEMESTER:III CLASS: II M. Com CA

S.NO	LECTURE DURATION PERIOD	TOPICS TO BE COVERED	SUPPORT MATERIAL/PAGE NOS
		UNIT-I	
1.	1	Definition of Software Engineering, Software characteristics and Application.	**Software engineering a practitioners approach , 20,245**
2.	1	Software myths, Software engineering- A layered technology and SDLC	**Software engineering a practitioners approach, 12**
3.	1	Software process models: Linear sequential model, prototyping model, RAD Model	**Software engineering a practitioners approach, 26**
4.	1	Evolutionary process models: Incremental process models and Spiral model.	**Software engineering a practitioners approach, 34**
5.	1	Component based ,4GT Process Models	**Software engineering a practitioners approach , 42,44**
6.	1	Maturity Models: CMM, CMMI, PCMM	**www.tutorialdspoint.com**
7.	1	PSP, TSP, Process patterns, process assessment	https://resources.sei.cmu.edu/library/
8.	1	Unified process: SEI CMM and ISO 9001. PSP and Six Sigma. Clean room technique.	**Software engineering a practitioners approach, 216**
9.	1	Recap of Unit I	
		UNIT-II	
1	1	The management spectrum, software quality, measurement and metrics	https://www.onlineclassnotes.com/2013/01/is-management-spectrum-describe-four-ps.html,

			Software engineering a practitioners approach, 193
2	1	Software project estimation, decomposition techniques	**Software engineering a practitioners approach, 124**
3	1	Empirical estimation models(COCOMO), the Make & Buy Decision	**Software engineering a practitioners approach, 133**
4	1	System models: Context Models, Behavioral models, Data models, Object models. Design process, Design quality and design model.	**Software engineering a practitioners approach, 311**
5	1	Fundamental issues in software design: Goodness of design, cohesions, coupling.	**Software engineering a practitioners approach, 324**
6	1	Function-oriented design and object – oriented concepts. Architectural styles and patterns	**Software engineering a practitioners approach, 373**
7	1	Architectural Design: Unified Modeling Language (UML), User interface design	**Software engineering a practitioners approach, 575**
8	1	Risk Analysis and management.	**Software engineering a practitioners approach, 145, 829**
9	1	Recap of Unit II	
UNIT-III			
1	1	S/W Requirements: Functional and non-functional requirements, User requirements, System requirements & SRS	**Software engineering a practitioners approach, 292**
2	1	S/W Metrices: Process Metrices, Project Metrices & Product Metrices	**Software engineering a practitioners approach, 74,80,507**
3	1	Testing Strategies: A strategic approach to software testing, Testing fundamentals	**Software engineering a practitioners approach, 477**
4	1	Test Case Design	**Software engineering a practitioners approach, 477**
5	1	Types of Testing: Black-Box Testing, White-Box Testing	**Software engineering a practitioners approach, 477**
6	1	Validation testing, System testing, The art of Debugging.Code walkthrough and reviews	**Software engineering a practitioners approach, 477**
7	1	Software Quality, Metrics for Analysis Model, Metrics for Design Model	**Software engineering a practitioners approach, 193**

8	1	Metrics for source code, Metrics for testing, Metrics for maintenance.	http://ecomputernotes.com/software-engineering/classification-of-software-metrics
9	1	Recap of Unit III	
UNIT-VI			
1	1	Snooping for information, Coping with complexity through teaming	https://www.coursehero.com/file/pnnhb9/A-Generic-view-of-process-Software-engineering-A-layered-technology-a-process/
2	1	Testing plan focus areas, Testing for recoverability, Planning for troubles	**Software engineering a practitioners approach, 477, 507**
3	1	Testing plan focus areas, Testing for recoverability, Planning for troubles	**Software engineering a practitioners approach, 477, 507**
4	1	Preparing for the tests: Software Reuse, Developing good test programs	**Software engineering a practitioners approach**
5	1	Data corruption, Tools, Test Execution	**Software engineering a practitioners approach**
6	1	Testing with a virtual computer, Simulation and Prototypes	**Software engineering a practitioners approach**
7	1	Managing the Test, Customer's role in testing	**Software engineering a practitioners approach**
8	1	Software maintenance issues and techniques. Software reuse, Client-Server software development	**Software engineering a practitioners approach**
9	1	Recap of Unit IV	
UNIT-V			
1	1	Software Reengineering, Reverse Engineering & Forward Engineering	**Software engineering a practitioners approach, 747**
2	1	Life Cycle Phases and Process artifacts, Restructuring	http://umlguide2.uw.hu/app02lev1sec2.html
3	1	Model based software architectures	http://www.pvpsiddhartha.ac.in/dep_it/lecture%20notes/SPM/unit3.pdf
4	1	Software process and Iteration workflows, Major and Minor milestones	sigc.edu/department/mca/studymet/SoftwareProjectManagment.pdf
5	1	Periodic status assessments, Process Planning, Project Control	Software Engineering, V. Anitha Moses, Lakshmi Publications.
6	1	process instrumentation: Seven core metrics, management indicators, quality indicators, life-cycle expectations	Software Engineering, V. Anitha Moses, Lakshmi Publications.

7	1	CCPDS-R Case, Study Future Software Project Management Practices	https://project-management-software.financesonline.com/future-project-management/
8	1	Future Software Project Management Practices	https://project-management-software.financesonline.com/future-project-management/
9	1	Recap of Unit V	
10	1	Review of Previous question paper	
11	1	Review of Previous question paper	
12	1	Review of Previous question paper	

TEXT BOOKS

1. ISO/IEC TR 19759:2015(en), Software Engineering — Guide to the software engineering body of knowledge (SWEBOK)
2. Software Engineering, V. Anitha Moses, Lakshmi Publications.
3. Software Engineering: A Practitioner's Approach. 4/e, Roger. S. Pressman, McGraw-Hill International Editions.

REFERENCES

1. Jessica Keyes. **Software Engineering** Handbook . ...
2. Ian Sommerville. **Software Engineering** (Seventh Edition). ...

KARPAGAM ACADEMY OF HIGHER EDUCATION

CLASS:IIM.Com(CA)

COURSE NAME: SOFTWARE MODELS AND ENGINEERING

COURSECODE:18CCP301

BATCH-2018-2020

UNIT-I

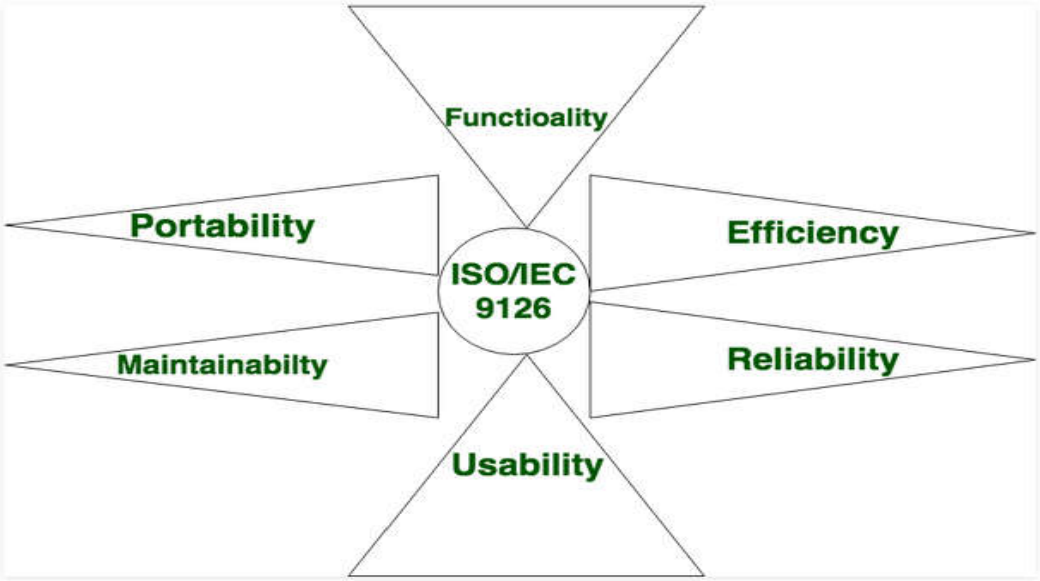
SYLLABUS

Fundamentals of Software Engineering and Process models : Definition, Software characteristics and Application. Software myths, Software engineering- A layered technology and SDLC. Software process models: Linear sequential model, prototyping model, RAD Model. Evolutionary process models: Incremental process models and Spiral model. Component based ,4GT. Maturity Models: CMM, CMMI, PCMM, PSP, TSP, Process patterns, process assessment. Unified process: SEI CMM and ISO 9001. PSP and Six Sigma. Clean room technique.

Software Engineering definition proposed by Fritz Bauer at the seminal conference on the subject still serves as a basis for discussion:

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

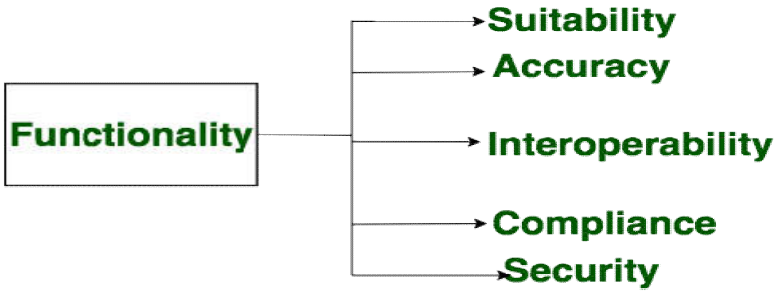
Software is defined as collection of computer programs, procedures, rules and data. Software Characteristics are classified into six major components:



These components are described below:

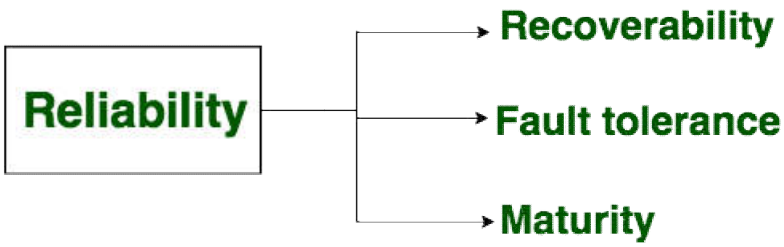
Functionality:

It refers to the degree of performance of the software against its intended purpose. Required functions are:



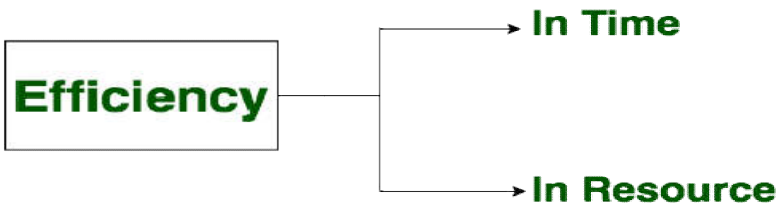
Reliability:

A set of attribute that bear on capability of software to maintain its level of performance under the given condition for a stated period of time. Required functions are:



- **Efficiency:**

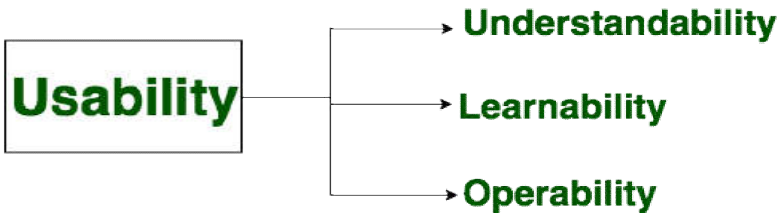
  It refers to the ability of the software to use system resources in the most effective and efficient manner. The software should make effective use of storage space and executive command as per desired timing requirement. Required functions are:



- **Usability:**

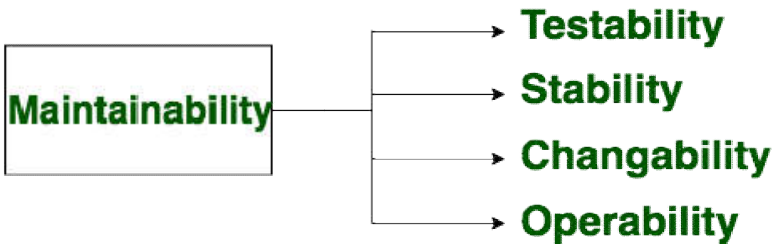
  It refers to the extent to which the software can be used with ease, the amount of effort or time required to learn how to use the software.

  Required functions are:



- **Maintainability:**

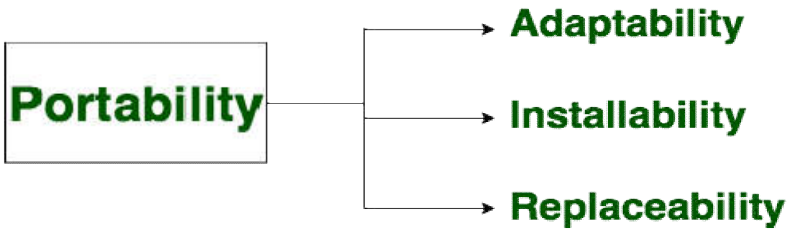
  It refers to the ease with which the modifications can be made in a software system to extend its functionality, improve its performance, or correct errors. Required functions are:



- **Portability:**

  A set of attribute that bear on the ability of software to be transferred from one environment to another, without or minimum changes.

  Required functions are:



Software myths—It is erroneous belief about software and the process that is used to build it—can be traced to the earliest days of computing. Myths have a number of attributes that make them insidious. For instance, they appear to be reasonablestatements of fact (sometimes containing elements of truth), they have an intuitive feel, and they are often promulgated by experienced practitioners who "know the score." Today, most knowledgeable software engineering professionals recognize myths for what they are—misleading attitudes that have caused serious problems for managers and practitioners alike. However, old attitudes and habits are difficult to modify, and remnants of software myths remain.

Management myths. Managers with software responsibility, like managers inmost disciplines, are often under pressure to maintain budgets, keep schedules fromslipping, and improve quality. Like a drowning person who grasps at a straw, a softwaremanager often grasps at belief in a software myth, if that belief will lessen thepressure (even temporarily).

Myth: We already have a book that's full of standards and procedures forbuilding software. Won't that provide my people with everything theyneed to know?Reality: Are softwarepractitioners aware of its existence? Does it reflect modernsoftware engineering

practice? Is it complete? Is itstreamlined to improve time-to-delivery while still maintaining afocus on quality? In many cases, the answer to all of these questionsis “no.”

Myth: If we get behind schedule, we can add more programmers and catch up(sometimes called the “Mongolian horde” concept).

Reality: Software development is not a mechanistic process like manufacturing.In the words of Brooks : “adding people to a late softwareproject makes it later.” At first, this statement may seemcounterintuitive. However, as new people are added, people whowere working must spend time educating the newcomers, therebyreducing the amount of time spent on productive developmenteffort. People can be added but only in a planned and coordinatedmanner.

Myth: If I decide to outsource the software project to a third party, I can justrelax and let that firm build it.

Reality: If an organization does not understand how to manage and controlsoftware projects internally, it will invariably struggle when it outsourcessoftware projects.

Customer myths. A customer who requests computer software may be a personat the next desk, a technical group down the hall, the marketing/sales department,or an outside company that has requested software under contract. In many cases,the customer believes myths about software because software managers and practitionersdo little to correct misinformation. Myths lead to false expectations (by thecustomer) and, ultimately, dissatisfaction with the developer.

Myth: A general statement of objectives is sufficient to begin writingprograms—we can fill in the details later.

Reality: Although a comprehensive and stable statement of requirements isnot always possible, an ambiguous “statement of objectives” is arecipe for disaster. Unambiguous requirements (usually derivediteratively) are developed only through effective and continuouscommunication between customer and developer.

Myth: Software requirements continually change, but change can be easilyaccommodated because software is flexible.

Reality: It is true that software requirements change, but the impact ofchange varies with the time at which it is introduced. When requirementschanges are requested early (before design or code has beenstarted), the cost impact is relatively small. However, as timepasses, the cost impact grows rapidly—resources have been committed,a design framework has been established, and change cancause upheaval that requires additional resources and major designmodification.

Practitioner's myths. Myths that are still believed by software practitioners havebeen fostered by over 50 years of programming culture. During the early days, programmingwas viewed as an art form. Old ways and attitudes die hard.

Myth: Once we write the program and get it to work, our job is done.Reality: Someone once said that "the sooner you begin 'writing code,' thelonger it'll take you to get done." Industry data indicate that between60 and 80 percent of all effort expended on software will be expendedafter it is delivered to the customer for the first time.

Myth: Until I get the program "running" I have no way of assessing its quality.
Reality: One of the most effective software quality assurance mechanismscan be applied from the inception of a project—the technical review.

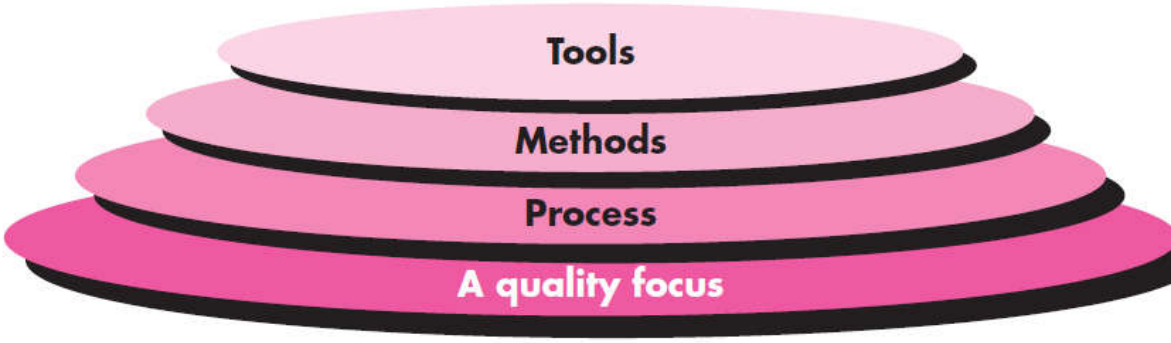
Myth: The only deliverable work product for a successful project is the workingprogram.
Reality: A working program is only one part of a software configuration thatincludes many elements. A variety of work products (e.g., models, documents, plans) provide a foundation for successful engineeringand, more important, guidance for software support.
Myth: Software engineering will make us create voluminous and unnecessarydocumentation and will invariably slow us down.
Reality: Software engineering is not about creating documents. It is aboutcreating a quality product. Better quality leads to reduced rework.And reduced rework results in faster delivery times.

Layered Technology

Software engineering is a layered technology. Referring to Figure 1.3, any engineering approach (including software engineering) must rest on an organizational commitment to quality. Total

quality management, Six Sigma, and similar philosophies foster a continuous process improvement culture, and it is this culture that ultimately leads to the development of increasingly more effective approaches to software engineering. The bedrock that supports software engineering is a quality focus. The foundation for software engineering is the process layer. The software engineeringprocess is the glue that holds the technology layers together and enables rational and timely development of computer software.

FIGURE 1.3
Software engineering layers



Software engineering is Divided into **4** layers:-

1. A quality Process :-

- Any engineering approach must rest on quality.
- The "Bed Rock" that supports software Engineering is Quality Focus.

2. Process :-

- Foundation for SE is the Process Layer
- SE process is the GLUE that holds all the technology layers together and enables the timely development of computer software.
- It forms the base for management control of software project.

3. Methods :-

- SE methods provide the "Technical Questions" for building Software.
- Methods contain a broad array of tasks that include communication requirement analysis, design modeling, program construction testing and support.

4. Tools :-

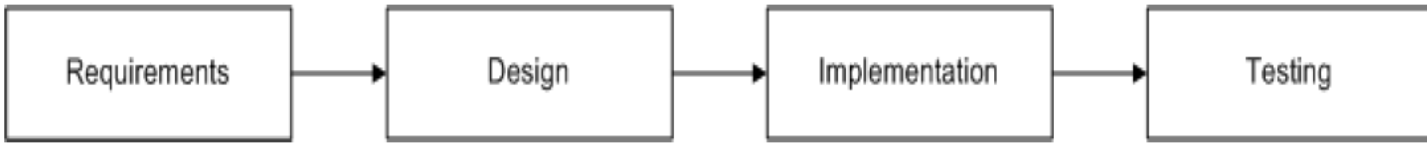
- SE tools provide automated or semi-automated support for the "Process" and the "Methods".
- Tools are integrated so that information created by one tool can be used by another.

A maturity level is a well-defined evolutionary plateau toward achieving a mature software process. Each maturity level provides a layer in the foundation for continuous process improvement.

Software Life Cycle

A Life Cycle shows how a living thing borns, grows, lives, and dies. The stages from birth to death.Software life cycle model is the stages of development that a software development goes through. The following figure shows the stages of software development.

Software life cycle models describe phases of the software cycle and the order in which those phases are executed. There are tons of models, and many companies adopt their own, but all have very similar patterns. The general, basic model is shown below:



Each phase produces deliverables required by the next phase in the life cycle. Requirements are translated into design. Code is produced during implementation that is driven by the design. Testing verifies the deliverable of the implementation phase against requirements.

A **Software Process** can be defined as set of activities, methods, practices and transformations which people employ to develop and maintain software and the associated products. The quality of a software product is essentially determined by the quality of the processes employed to develop and maintain it.

The Linear Sequential Model

This is a software process model that involves a systematic progression through **analysis**, **design**, **coding**, **testing** and **maintenance** phases. It is also referred to as the "**waterfall model**".

Also known as the classic life cycle or waterfall model, it suggests a systematic, sequential approach to software development. Problems with this approach are:

• Real projects rarely follow the sequential flow and changes can cause confusion.

• This model has difficulty accommodating requirements change

• The customer will not see a working version until the project is nearly complete

• Developers are often blocked unnecessarily, due to previous tasks not being done

The Prototyping Model

Advantages:

- Easy and quick to identify customer requirements

- Customers can validate the prototype at the earlier stage and provide their inputs and feedback

- Good to deal with the following cases:

1. Customer cannot provide the detailed requirements
2. Very complicated system-user interactions
3. Use new technologies, hardware and algorithms
4. Develop new domain application systems

Problems:

-The prototype can serve as **—the first system**.

-Developers usually attempt to develop the product based on the prototype.

-Developers often make implementation compromises in order to get a prototyping that is working quickly.

-Customers may be unaware that the prototype is not a product, which is held with.

The RAD Model

Rapid Application Development (RAD) is a linear sequential software development process model that emphasizes an extremely short development cycle.

- A ―high-speed‖ adaptation of linear sequential model

- Component-based construction

- Effective when requirements are well understood and project scope is constrained.

Advantages:

- Short development time

- Cost reduction due to software reuse and component-based construction

Problems:

- For large, but scalable projects, RAD requires sufficient resources.

- RAD requires developers and customers who are committed to the schedule.
- Constructed software is project-specific, and may not be well modularized.
- Its quality depends on the quality of existing components.
- Not appropriate projects with high technical risk and new technologies.

Incremental Process Models

There are many situations in which initial software requirements are reasonably well defined, but the overall scope of the development effort precludes a purely linear process. In addition, there may be a compelling need to provide a limited set of software functionality to users quickly and then refine and expand on that functionalityin later software releases. In such cases, you can choose a process model that is designed to produce the software in increments. The incremental model combines elements of linear and parallel process flows

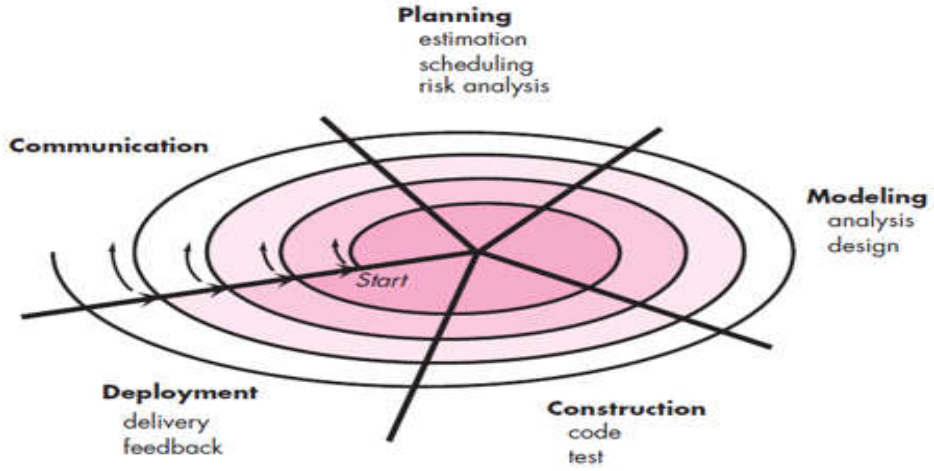
Incremental model applies linear sequences in a staggered fashion as calendar time progresses. Each linear sequence produces deliverable "increments" of the software in a manner that is similar to the increments produced by an evolutionary process flow.For example, word-processing software developed using the incremental paradigm might deliver basic file management, editing, and document production functionsin the first increment; more sophisticated editing and document production capabilities in the second increment; spelling and grammar checking in the third increment; and advanced page layout capability in the fourth increment. It should be noted that the process flow for any increment can incorporate the prototyping paradigm.

When an incremental model is used, the first increment is often a core product. That is, basic requirements are addressed but many supplementary features (some known, others unknown) remain undelivered. The core product is used by the customer (or undergoes detailed evaluation).

The Spiral Model.

Originally proposed by Barry Boehm, the spiral model is an evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the waterfall model. It providesthe potential for rapid development of increasingly more complete versions of the software. Boehm describes the model in the following manner: The spiral development model is a risk-driven process model generator that is used to guide multi-stakeholder concurrent engineering of software intensive systems. It has two main distinguishing features. One is a cyclic approach for incrementally growing a system's degree of definition and

implementation while decreasing its degree of risk. The other is a set of anchor point milestones for ensuring stakeholder commitment to 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 model or prototype. During later iterations, increasingly more complete versions of the engineered system are produced.



A Typical Spiral Model

Component-based development model

Commercial off-the-shelf (COTS) software components, developed by vendors who offer them as products, provide targeted functionality with well-defined interfaces that enable the component to be integrated into the software that is to be built. The component-based development model incorporates many of the characteristics of the spiral model. It is evolutionary in nature, demanding an iterative approach to the creation of software. However, the component-based development model constructs applications from prepackaged software components. Modeling and construction activities begin with the identification of candidate components. These components can be designed as either conventional software modules or object-oriented classes or packages of classes. Regardless of the technology that is used to create the components, the component-based development

model incorporates the following steps (implemented using an evolutionary approach):

1. Available component-based products are researched and evaluated for theapplication domain in question.

2. Component integration issues are considered.

3. A software architecture is designed to accommodate the components.

4. Components are integrated into the architecture.

5. Comprehensive testing is conducted to ensure proper functionality.

The component-based development model leads to software reuse, and reusability provides software engineers with a number of measurable benefits. Your software engineering team can achieve a reduction in development cycle time as well as a reduction in project cost if component reuse becomes part of your culture.

4GT Process Model

4GT begins with from "Requirement Gathering" this process go through the customer, the customer go illustrate the requirements. The customer could actually describe the requirements and these would be directly translated into an operational prototype. If the product is a smaller product this process may be possible to move directly from requirements gathering step to implementation using a non-procedural fourth generation language (4GL), for lager products this procedure may little hard, therefor it's necessary to use the design strategy in 4GT.When it comes to large projects, the design phase it is crucial to avoid poor quality, poor maintainability. To transform a 4GL implementation into a product, the developer must conduct through testing, develop meaningful documentation, and perform all other solution integration activities. The 4GT developed software must be built in a manner that enables maintenance to be performed expeditiously. There are some merits to summarize the current features of 4GT approach. In the 4GL implementation the code can be generated based on some specification. The 4GT developed software must be built in a manner that enables maintenance to be performed expeditiously. There are some merits to summarize the current features of 4GT approach. The use of 4GT is a viable approach for many different application areas coupled with computer- aided software engineering tools and code generators, 4GT offers a credible solution to many software problem.

Benefits of the 4GT

Flexible: The Fourth Generation applications are Modifiable by Design, which means they are designed from the beginning to accommodate change. They are easily modifiable, either by you, the customer, or by your Fourth Generation Authorized reseller to your specifications.

Scalable: The Fourth Generation applications are Modifiable by Design, which means they are designed from the beginning to accommodate change.They are easily modifiable, either by you, the customer, or by your Fourth Generation Authorized reseller to your specifications.

Total Data Access: Your data represents your company's greatest single asset. The worth of that asset, however, is directly related to your ability to record it and access it.

The Fourth Generation Technique (4GT) is based on NPL that is the Non-Procedural Language techniques. Depending upon the specifications made the 4GT move towards uses various tools for the automatic generation of source codes. It is the very important tool which make use of the non-procedural language for Report generation, Database query, Manipulation of data, Interaction of screen, Definition, Generation of code, Spread Sheet capabilities, and High level graphical capacity etc. 4GT begins with a requirement-gathering stage. The customer would illustrate requirements and these would be directly converted into an unworkable operational prototype. For small applications, it may be possible to move directly from requirements gathering step to implementation using a non-procedural fourth generation language (4GL), however for large application it is necessary to develop a design strategy for the system even if a 4GL is to be used. Implementation using a 4GT enables the software developer to represent desired result in a manner that leads to automatic generation of code to create those results, obviously, data structure with relevant information must exist and be readily accessible by the 4GL. To transform a 4GL implementation into a product, the developer must conduct through testing, develop meaningful documentation, and perform all other solution integration activities. The 4GT developed software must be built in a manner that enables maintenance to be performed expeditiously. There are some merits to summarize the current features of 4GT approach. The use of 4GT is a viable approach for many different application areas coupled with computer- aided software engineering tools and code generators, 4GT offers a credible solution to many software problem. Data collected from companies that use 4Gt indicates that the time required to produce software is greatly reduced for small and intermediate application is also reduced. However the use of 4GT for large software development efforts demands as much or more analysis design and testing to achieve substantial timesaving that result from the elimination of coding.

Compatibility Maturity Model -CMM

Maturity level 1 _Initial

organizations often produce products and services that work; however, they frequently exceed the budget and schedule of their projects.Maturity level 1 organizations are characterized by a tendency to over commit, abandon processes in the time of crisis, and not be able to repeat their past successes.

Maturity Level 2 - Managed

At maturity level 2, an organization has achieved all the specific and generic goals of the maturity level 2 process areas. In other words, the projects of the organization have ensured that requirements are managed and that processes are planned, performed, measured, and controlled.The process discipline reflected by maturity level 2 helps to ensure that existing practices are retained during times of stress. When these practices are in place, projects are performed and managed according to their documented plans.At maturity level 2, requirements, processes, work products, and services are managed. The status of the work products and the delivery of services are visible to management at defined points.Commitments are established among relevant stakeholders and are revised as needed. Work products are reviewed with stakeholders and are controlled.The work products and services satisfy their specified requirements, standards, and objectives.

Maturity Level 3 - Defined

At maturity level 3, an organization has achieved all the specific and generic goals of the process areas assigned to maturity levels 2 and 3.At maturity level 3, processes are well characterized and understood, and are described in standards, procedures, tools, and methods.A critical distinction between maturity level 2 and maturity level 3 is the scope of standards, process descriptions, and procedures. At maturity level 2, the standards, process descriptions, and procedures may be quite different in each specific instance of the process (for example, on a particular project). At maturity level 3, the standards, process descriptions, and procedures for a project are tailored from the organization's set of standard processes to suit a particular project or organizational unit. The organization's set of standard processes includes the processes addressed at maturity level 2 and maturity level 3. As a result, the processes that are performed across the organization are consistent except for the differences allowed by the tailoring guidelines.Another

critical distinction is that at maturity level 3, processes are typically described in more detail and more rigorously than at maturity level 2. At maturity level 3, processes are managed more proactively using an understanding of the interrelationships of the process activities and detailed measures of the process, its work products, and its services.

Maturity Level 4 - Quantitatively managed

At maturity level 4, an organization has achieved all the specific goals of the process areas assigned to maturity levels 2, 3, and 4 and the generic goals assigned to maturity levels 2 and 3.At maturity level 4 Sub processes are selected that significantly contribute to overall process performance. These selected sub processes are controlled using statistical and other quantitative techniques.Quantitative objectives for quality and process performance are established and used as criteria in managing processes. Quantitative objectives are based on the needs of the customer, end users, organization, and process implementers. Quality and process performance are understood in statistical terms and are managed throughout the life of the processes.

Maturity Level 5 - Optimizing

At maturity level 5, an organization has achieved all the specific goals of the process areas assigned to maturity levels 2, 3, 4, and 5 and the generic goals assigned to maturity levels 2 and 3.Processes are continually improved based on a quantitative understanding of the common causes of variation inherent in processes.Maturity level 5 focuses on continually improving process performance through both incremental and innovative technological improvements.Quantitative process-improvement objectives for the organization are established, continually revised to reflect changing business objectives, and used as criteria in managing process improvement.

Capability Maturity Model Integration - CMMI

Capability Maturity Model Integration (CMMI) is a process level improvement training and appraisal program. Administered by the CMMI Institute, a subsidiary of ISACA, it was developed at Carnegie Mellon University (CMU). It is required by many United States Department of Defense (DoD) and U.S. Government contracts, especially in software development. CMU claims CMMI can be used to guide process improvement across a project, division, or an entire organization. CMMI defines the following maturity levels for processes: Initial, Repeatable, Defined, Quantitatively Managed, and Optimizing.

CMMI Model

1) Initial: The software process is characterized as ad hoc, and occasionally even chaotic. Few processes are defined, and success depends on individual effort and heroics.

2) Repeatable: Basic project management processes are established to track cost, schedule, and functionality. The necessary process discipline is in place to repeat earlier successes on projects with similar applications.

3) Defined: The software process for both management and engineering activities is documented, standardized, and integrated into a standard software process for the organization. All projects use an approved, tailored version of the organization's standard software process for developing and maintaining software.

4) Managed: Detailed measures of the software process and product quality are collected. Both the software process and products are quantitatively understood and controlled.

5) Optimizing: Continuous process improvement is enabled by quantitative feedback from the process and from piloting innovative ideas and technologies.

People Capability Maturity Model (PCMM)

People Capability Maturity Model (PCMM) is a maturity framework that focuses on continuously improving the management and development of the human assets of a software or

information systems organization. PCMM can be perceived as the application of the principles of Capability Maturity Model to human assets of a software organization. It describes an evolutionary improvement path from ad hoc, inconsistently performed practices, to a mature, disciplined, and continuously improving development of the knowledge, skills, and motivation of the workforce. Although the focus in People CMM is on software or information system organizations, the processes and practices are applicable for any organization that aims to improve the capability of its workforce. PCMM will be guiding and effective particularly for organizations whose core processes are knowledge intensive.The primary objective of the People Capability Maturity Model is to improve the capability of the entire workforce. This can be defined as the level of knowledge, skills, and process abilities available for performing an organization's current and future business activities.

10 Principles of People Capability Maturity Model (PCMM)

The People Capability Maturity Model describes an evolutionary improvement path from ad hoc, inconsistently performed workforce practices, to a mature infrastructure of practices for continuously elevating workforce capability. The philosophy implicit the PCMM can be summarized in ten principles.In mature organizations, workforce capability is directly related to business performance.

Workforce capability is a competitive issue and a source of strategic advantage. Workforce capability must be defined in relation to the organization's strategic business objectives.

Knowledge-intense work shifts the focus from job elements to workforce competencies. Capability can be measured and improved at multiple levels, including individuals, workgroups, workforce competencies, and the organization.

An organization should invest in improving the capability of those workforce competencies that are critical to its core competency as a business.

Operational management is responsible for the capability of the workforce.

The improvement of workforce capability can be pursued as a process composed from proven practices and procedures.

The organization is responsible for providing improvement opportunities, while individuals are responsible for taking advantage of them.

Since technologies and organizational forms evolve rapidly, organizations must continually evolve their workforce practices and develop new workforce competencies. The People Capability Maturity Model (People CMM) is a roadmap for implementing workforce practices that continuously improve the capability of an organization's workforce. Since an organization cannot implement all of the best workforce practices in an afternoon, the People CMM introduces them in stages. Each progressive level of the People CMM produces a unique transformation in the organization's culture by equipping it with more powerful practices for attracting, developing, organizing, motivating, and retaining its workforce. Thus, the People CMM establishes an integrated system of workforce practices that matures through increasing alignment with the organization's business objectives, performance, and changing needs.

Although the People CMM has been designed primarily for application in knowledge intense organizations, with appropriate tailoring it can be applied in almost any organizational setting. The People CMM's primary objective is to improve the capability of the workforce. Workforce capability can be defined as the level of knowledge, skills, and process abilities available for performing an organization's business activities.

Personal Software Process (PSP)

Every developer uses some process to build computer software. The process may be haphazard or ad hoc; may change on a daily basis; may not be efficient, effective, or even successful; but a "process" does exist. Watts Humphrey [Hum97] suggests that in order to change an ineffective personal process, an individual must move through four phases, each requiring training and careful instrumentation. The Personal Software Process (PSP) emphasizes personal measurement of both the work product that is produced and the resultant quality of the work product. In addition PSP makes the practitioner responsible for project planning (e.g., estimating and scheduling) and empowers the practitioner to control the quality of all software work products that are developed. The PSP model defines five framework activities:

Planning.

This activity isolates requirements and develops both size and resource estimates. In addition, a defect estimate (the number of defects projected for the work) is made. All metrics are recorded on worksheets or templates. Finally, development tasks are identified and a project schedule is

created.

High-level design. External specifications for each component to be constructed are developed and a component design is created. Prototypes are built when uncertainty exists. All issues are recorded and tracked.

High-level design review. Formal verification methods are applied to uncover errors in the design. Metrics are maintained for all important tasks and work results.

Development. The component-level design is refined and reviewed. Code is generated, reviewed, compiled, and tested. Metrics are maintained for all important tasks and work results.

Postmortem. Using the measures and metrics collected (this is a substantial amount of data that should be analyzed statistically), the effectiveness of the process is determined. Measures and metrics should provide guidance formodifying the process to improve its effectiveness.

Team Software Process (TSP)

Because many industry-grade software projects are addressed by a team of practitioners, Watts Humphrey extended the lessons learned from the introduction of PSP and proposed a Team Software Process (TSP). The goal of TSP is to build a "selfdirected" project team that organizes itself to produce high-quality software.

Humphrey defines the following objectives for TSP:

• Build self-directed teams that plan and track their work, establish goals, andown their processes and plans. These can be pure software teams or integratedproduct teams (IPTs) of 3 to about 20 engineers.

• Show managers how to coach and motivate their teams and how to helpthem sustain peak performance.

• Accelerate software process improvement by making Level 5behavior normal and expected.

• Provide improvement guidance to high-maturity organizations.

• Facilitate university teaching of industrial-grade team skills.

A self-directed team has a consistent understanding of its overall goals and objectives; defines roles and responsibilities for each team member; tracks quantitativeproject data (about productivity and quality); identifies a team process that is appropriate for the project and a strategy for implementing the process; defines local standards that are applicable to the team's software engineering work; continually assesses risk and reacts to it; and tracks, manages, and reports project status.

TSP defines the following framework activities: project launch, high-level design, implementation, integration and test, and postmortem. Like their counterparts in PSP (note that terminology is somewhat different), these activities enable the team to plan, design, and construct software in a disciplined manner while at the same time quantitatively measuring the process and the product. The postmortem sets the stage for process improvements.

PROCESS PATTERNS

Every software team encounters problems as it moves through the software process. It would be useful if proven solutions to these problems were readily available to the team so that the problems could be addressed and resolved quickly. A *process pattern* describes a process-related problem that is encountered during software engineering work, identifies the environment in which the problem has been encountered, and suggests one or more proven solutions to the problem. Stated in more general terms, a process pattern provides you with a template—a consistent method for describing problem solutions within the context of the software process. By combining patterns, a software team can solve problems and construct a process that best meets the needs of a project.

Patterns can be defined at any level of abstraction. In some cases, a pattern might be used to describe a problem (and solution) associated with a complete processmodel (e.g., prototyping). In other situations, patterns can be used to describe a problem (and solution) associated with a framework activity (e.g., **planning**) or an action within a framework activity (e.g., project estimating).

Ambler has proposed a template for describing a process pattern:

Pattern Name. The pattern is given a meaningful name describing it within the context of the software process (e.g., **TechnicalReviews**).

Forces. The environment in which the pattern is encountered and the issues that make the problem visible and may affect its solution.

Type. Ambler suggests three types of patterns:

1. Stage pattern—defines a problem associated with a framework activity for the process. Since a framework activity encompasses multiple actions andwork tasks, a stage pattern incorporates multiple task patterns (see the following) that are relevant to the stage (framework activity). An example of astage pattern might be EstablishingCommunication. This pattern would incorporate the task pattern RequirementsGathering and others.

2. Task pattern—defines a problem associated with a software engineering action or work task and relevant to successful software engineeringpractice (e.g., RequirementsGathering is a task pattern).

3. Phase pattern—define the sequence of framework activities that occurs within the process, even when the overall flow of activities is iterativein nature. An example of a phase pattern might be **SpiralModel** or **Prototyping.**

PROCESS ASSESSMENT

The existence of a software process is no guarantee that software will be delivered on time, that it will meet the customer's needs, or that it will exhibit the technical characteristics that will lead to long-term quality characteristics. Process patterns must be coupled with solid software engineering practice

(Part 2 of this book). In addition, the process itself can be assessed to ensure that it meets a set of basic process criteria that have been shown to be essential for a successful software engineering.. A number of different approaches to software process assessment and improvement have been proposed over the past few decades:

Standard CMMI Assessment Method for Process Improvement

(SCAMPI)—provides a five-step process assessment model that incorporates five phases: initiating, diagnosing, establishing, acting, and learning. The

SCAMPI method uses the SEI CMMI as the basis for assessment.

CMM-Based Appraisal for Internal Process Improvement (CBA IPI) — provides a diagnostic technique for assessing the relative maturity of asoftware organization; uses the SEI CMM as the basis for the assessment.

SPICE (ISO/IEC15504)—a standard that defines a set of requirements for software process assessment. The intent of the standard is to assist organizations

in developing an objective evaluation of the efficacy of any defined software process .

ISO 9001:2000 for Software—a generic standard that applies to any organization that wants to improve the overall quality of the products, systems,

or services that it provides. Therefore, the standard is directly applicable to software organizations and companies

Six Sigma for Software Engineering

Six Sigma is the most widely used strategy for statistical quality assurance in industry today. Originally popularized by Motorola in the 1980s, the Six Sigma strategy "is a rigorous and disciplined methodology that uses data and statistical analysis to measure and improve a company's operational performance by identifying and eliminating defects' in manufacturing and service-related processes". The term Six Sigma is derived from six standard deviations—3.4 instances (defects) per million occurrences—implying an extremely high quality standard. The Six Sigma methodology defines three core steps:

• *Define* customer requirements and deliverables and project goals via welldefined methods of customer communication.

• *Measure* the existing process and its output to determine current quality performance (collect defect metrics).

• *Analyze* defect metrics and determine the vital few causes.

If an existing software process is in place, but improvement is required, Six Sigmasuggests two additional steps:

• *Improve* the process by eliminating the root causes of defects.

• *Control* the process to ensure that future work does not reintroduce the causes of defects.

Clean room technique (clean room design)

The clean room technique is a process in which a new product is developed by **reverse engineering** an existing product, and then the new product is designed in such a way that patent or **copyright** infringement is avoided. The clean room technique is also known as clean room design. (Sometimes the words "clean room" are merged into the single word, "cleanroom.") Sometimes this process is called the **Chinese wall** method, because the intent is to place a demonstrable intellectual barrier between the reverse engineering process and the development of the new product.

The use of the clean room technique can be compared, in some respects, with the fair use of copyrighted publications in order to compile a new document. For example, a new book about Linux can be authored on the basis of information obtained by researching existing books, articles, white papers, and Web sites. This does not necessarily constitute copyright infringement, even though other books on Linux already exist, and even if the new book contains essentially the same information as the existing publications. However, this is the case only as long as passages from the existing works are not copied verbatim or nearly verbatim,

and as long as the new work does not have substantially the same structure as any of the existing works.

Use of the clean room technique puts engineers and enterprises in a legal gray area. If the owner of the original copyright or patent can demonstrate that the development of a new product was done by means of reverse engineering and is not significantly different from the existing product, a lawsuit may result. Any attempt to reverse engineer an existing product, and then create a new product based on the results of the reverse engineering process, should be undertaken only with the advice of a reputable attorney who is experienced in copyright infringement and reverse engineering issues**.**

UNIT II

KARPAGAM ACADEMY OF HIGHER EDUCATION

CLASS:IIM.Com(CA)

COURSE NAME: SOFTWARE MODELS AND ENGINEERING

COURSECODE:18CCP301

BATCH-2018-2020

Managing Software Projects & Design Engineering: The management spectrum, software quality, measurement and metrics. Software project estimation, decomposition techniques. Empirical estimation models (COCOMO), the Make & Buy Decision. System models: Context Models, Behavioral models, Data models, Object models. Design process, Design quality and design model. Fundamental issues in software design: Goodness of design, cohesions, coupling. Function-oriented design and object – oriented concepts. Architectural styles and patterns, Architectural Design: Unified Modeling Language (UML), User interface design. Risk Analysis and management.

MANAGEMENT SPECTRUM

Effective software project management focuses on the **four P's: people, product, process, and project**. The order is not arbitrary. The manager who forgets that software engineering work is an intensely human endeavor will never have success in project management. A manager who fails to encourage comprehensive stakeholder communication early in the evolution of a product risks building an elegant solution for the wrong problem. The manager who pays little attention to the process runs the risk of inserting competent technical methods and tools into a vacuum. The manager who embarks without a solid project plan jeopardizes the success of the project.

ThePeople

The cultivation of motivated, highly skilled software people has been discussed since the 1960s. In fact, the "people factor" is so important that the Software Engineering Institute has developed

a *People Capability Maturity Model* (People-CMM), in recognition of the fact that "every organization needs to continually improve its ability to attract, develop, motivate, organize, and retain the workforce needed to accomplish its strategic business objectives". The people capability maturity model defines the following key practice areas for software people: staffing, communication and coordination, work environment, performance management, training, compensation, competency analysis and development, career development, workgroup development, team/culture development, and others. Organizations that achieve high levels of People-CMM maturity have a higher likelihood of implementing effective software project management practices. The People-CMM is a companion to the *Software Capability Maturity Model– Integration* that guides organizations in the creation of a mature software process.

The Product

Before a project can be planned, product objectives and scope should be established, alternative solutions should be considered, and technical and management constraints should be identified. Without this information, it is impossible to define reasonable (and accurate) estimates of the cost, an effective assessment of risk, a realistic breakdown of project tasks, or a manageable project schedule that provides a meaningful indication of progress. As a software developer, you and other stakeholders must meet to define product objectives and scope. In many cases, this activity begins as part of the system engineering or business process engineering and continues as the first step in software requirements engineering. Objectives identify the overall goals for the product (from the stakeholders' points of view) without considering how these goals will be achieved. Scope identifies the primary data, functions, and behaviors that characterize the product, and more important, attempts to bound these characteristics in a quantitative manner. Once the product objectives and scope are understood, alternative solutions are considered. Although very little detail is discussed, the alternatives enable managers and practitioners to select a "best" approach, given the constraints imposed by delivery deadlines, budgetary restrictions, personnel availability, technical interfaces, and myriad other factors.

The Process

A software process provides the framework from which a comprehensive plan for software development can be established. A small number of framework activities are applicable to all software projects, regardless of their size or complexity. A number of different task sets—tasks, milestones, work products, and quality assurance points—enable the framework activities to be

adapted to the characteristics of the software project and the requirements of the project team. Finally, umbrella activities—such as software quality assurance, software configuration management, and measurement—overlay the process model. Umbrella activities are independent of any one framework activity and occur throughout the process.

The Project

We conduct planned and controlled software projects for one primary reason—it is the only known way to manage complexity. And yet, software teams still struggle. In a study of 250 large software projects between 1998 and 2004, Capers Jones found that "about 25 were deemed successful in that they achieved their schedule, cost, and quality objectives. About 50 had delays or overruns below 35 percent, while about 175 experienced major delays and overruns, or were terminated without completion." Although the success rate for present-day software projects may have improved somewhat, our project failure rate remains much higher than it should be. To avoid project failure, a software project manager and the software engineers who build the product must avoid a set of common warning signs, understand the critical success factors that lead to good project management, and develop a commonsense approach for planning, monitoring, and controlling the project.

Software quality, measurement and metrics

Software quality assurance is composed of a variety of tasks associated with two different constituencies—the software engineers who do technical work and an SQA group that has responsibility for quality assurance planning, oversight, record keeping, analysis, and reporting.

Software engineers address quality (and perform quality control activities) by applying solid technical methods and measures, conducting technical reviews, and performing well-planned software testing.

Measurements in the physical world can be categorized in two ways: direct measures (e.g., the length of a bolt) and indirect measures (e.g., the "quality" of bolts produced, measured by counting rejects). Software metrics can be categorized similarly.

Direct measures of the software process include cost and effort applied. Direct measures of the product include lines of code (LOC) produced, execution speed, memory size, and defects reported over some set period of time. Indirect measures of the product include functionality, quality, complexity, efficiency, reliability, maintainability, and many other "–abilities".

The cost and effort required to build software, the number of lines of code produced, and other direct measures are relatively easy to collect, as long as specific conventions for measurement are established in advance.

However, the quality and functionality of software or its efficiency or maintainability are more difficult to assess and can be measured only indirectly. The software metrics domain can be partitioned into process, project, and product metrics. Project metrics are then consolidated to create process metrics that are public to the software organization as a whole. But how does an organization combine metrics that come from different individuals or projects? To illustrate, consider a simple example. Individuals on two different project teams record and categorize all errors that they find during the software process. Individual measures are then combined to develop team measures. Team A found 342 errors during the software process prior to release. Team B found 184 errors. All other things being equal, which team is more effective in uncovering errors throughout the process? Because you do not know the size or complexity of the projects, you cannot answer this question. However, if the measures are normalized, it is possible to create software metrics that enable comparison to broader organizational averages.

Size-Oriented Metrics

Size-oriented software metrics are derived by normalizing quality and/or productivity measures by considering the size of the software that has been produced. If a software organization maintains simple records, a table of size-oriented measures can be created.

Function-Oriented Metrics

Function-oriented software metrics use a measure of the functionality delivered by the application as a normalization value. The most widely used function-oriented metric is the function point (FP). Computation of the function point is based on characteristics of the software's information domain and complexity.

The function point, like the LOC measure, is controversial. Proponents claim that FP is programming language independent, making it ideal for applications using conventional and nonprocedural languages, and that it is based on data that are more likely to be known early in the evolution of a project, making FP more attractive as an estimation approach. Opponents claim that the method requires some "sleight of hand" in that computation is based on subjective

rather than objective data, that counts of the information domain (and other dimensions) can be difficult to collect after the fact, and that FP has no direct physical meaning—it's just a number.

Object-Oriented Metrics

Conventional software project metrics (LOC or FP) can be used to estimate object orientedsoftware projects. However, these metrics do not provide enough granularity for the schedule and effort adjustments that are required as you iterate through an evolutionary or incremental process.

Lorenz and Kidd suggest the followingset of metrics for Object Oriented projects:

Number of scenario scripts:

A scenario script is a detailed sequence of steps that describe theinteraction between the user and the application. Each script is organized into triplets of the form

{initiator, action, participant}

where initiator is the object that requests some service (that initiates a message), action is the result of the request, and participant is the server object that satisfies the request. The number of scenario scripts is directly correlated to the size of the application and to the number of test cases that must be developed to exercise the system once it is constructed.

Number of key classes: Key classes are the "highly independent components" that are defined early in object-oriented analysis . Because key classes are central to the problem domain, the number of such classes is an indication of the amount of effort required to develop the software and also an indicationof the potential amount of reuse to be applied during system development.

Number of support classes: Support classes are required to implement the system but are not immediately related to the problem domain. Examples might be user interface (GUI) classes, database access and manipulation classes, and computation classes. In addition, support classes can be developed for each of the key classes. Support classes are defined iteratively throughout an evolutionary process. The number of support classes is an indication of the amount of effort required to develop the software and also an indication of the potential amount of reuse to be applied during system development.

Average number of support classes per key class: In general, key classes areknown early in the project. Support classes are defined throughout. If the average number of support classes per key class were known for a given problem domain, estimating (based on total number of classes) would be greatly simplified. Lorenz and Kidd suggest that applications with a GUI have between two and three times the number of support classes as key classes. Non-GUI applications have between one and two times the number of support classes as key classes.

Number of subsystems: A subsystem is an aggregation of classes that support a function that is visible to the end user of a system. Once subsystems are identified, it is easier to lay out a reasonable schedule in which work on subsystems is partitioned among project staff.

Use-Case–Oriented Metrics: Use cases are used widely as a method for describing customer-level or business domain requirements that imply software features and functions. It would seem reasonable to use the use case as a normalization measure similar to LOC or FP.

SOFTWARE PROJECT ESTIMATION

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

1. Delay estimation until late in the project (obviously, we can achieve 100 percentaccurate estimates after the project is complete!).
2. Base estimates on similar projects that have already been completed.
3. Use relatively simple decomposition techniques to generate project cost andeffort estimates.
4. Use one or more empirical models for software cost and effort estimation.

Unfortunately, the first option, however attractive, is not practical. Cost estimatesmust be provided up-front. However, you should recognize that the longer you wait,the more you know, and the more you know, the less likely you are to make seriouserrors in your estimates.The second option can work reasonably well, if the current project is quite similarto past efforts and other project influences (e.g., the customer, business conditions,the software engineering environment, deadlines) are roughly equivalent. Unfortunately,past experience has not always been a good indicator of future results.The remaining options are viable approaches to software

project estimation.Ideally, the techniques noted for each option should be applied in tandem; each usedas a cross-check for the other.

Decomposition techniques

Software project estimation is a form of problem solving, and in most cases, the problem to be solved (i.e., developing a cost and effort estimate for a software project) is too complex to be considered in one piece. For this reason, you should decompose the problem, recharacterizing it as a set of smaller (and hopefully, more manageable) problems.But before an estimate can be made, you must understand the scope of the software to be built and generate an estimate of its "size."

Software Sizing

The accuracy of a software project estimate is predicated on a number of things:

(1) the degree to which you have properly estimated the size of the product to be built;

(2) the ability to translate the size estimate into human effort, calendar time, and dollars (a function of the availability of reliable software metrics from past projects);

(3) the degree to which the project plan reflects the abilities of the software team; and

(4) the stability of product requirements and the environment that supports the software engineering effort.

Putnam and Myers suggest four different approaches to the sizing problem:

• "Fuzzy logic" sizing. This approach uses the approximate reasoning techniques that are the cornerstone of fuzzy logic. To apply this approach, the planner must identify the type of application, establish its magnitude on a qualitative scale, and then refine the magnitude within the original range.

Function point sizing. The planner develops estimates of the information domain characteristics

Standard component sizing. Software is composed of a number of different "standard components" that are generic to a particular application area. For example, the standard components for an information system are subsystems, modules, screens, reports, interactive programs, batch programs, files, LOC, and object-level instructions. The project planner estimates the number of occurrences of each standard component and then uses historical project data to estimate the delivered size per standard component.

Change sizing. This approach is used when a project encompasses the use of existing software that must be modified in some way as part of a project. The planner estimates the number and type (e.g., reuse, adding code, changing code, deleting code) of modifications that must be accomplished.

Problem-Based Estimation

LOC and FP data are used in two ways during software project estimation:

(1) as estimation variables to "size" each element of the software and

(2) as baseline metrics collected from past projects and used in conjunction with estimation variables to develop cost and effort projections.

LOC and FP estimation are distinct estimation techniques. Yet both have a number of characteristics in common. You begin with a bounded statement of software scope and from this statement attempt to decompose the statement of scope into problem functions that can each be estimated individually. LOC or FP (the estimation variable) is then estimated for each function. Alternatively, you may choose another component for sizing, such as classes or objects, changes, or business processes affected.

The LOC and FP estimation techniques differ in the level of detail required for decomposition and the target of the partitioning. When LOC is used as the estimation variable, decomposition is absolutely essential and is often taken to considerable levels of detail. The greater the degree of partitioning, the more likely reasonably accurate estimates of LOC can be developed. For FP estimates, decomposition works differently. Rather than focusing on function, each of the information domain characteristics—inputs, outputs, data files, inquiries, and external interfaces are estimated. The resultant estimates can then be used to derive an FP value that can be tied to past data and used to generate an estimate.

Regardless of the estimation variable that is used, you should begin by estimating a range of values for each function or information domain value. Using historical data or (when all else fails) intuition, estimate an optimistic, most likely, and pessimistic size value for each function or count for each information domain value. An implicit indication of the degree of uncertainty is provided when a range of values is specified. A three-point or expected value can then be computed. The expected value for the estimation variable (size) S can be computed as a weighted average of the optimistic (s_{opt}), most likely (s_m), and pessimistic (s_{pess}) estimates.

For example,

$$S = \frac{s_{opt} + 4s_m + s_{pess}}{6}$$

COCOMO - COnstructiveCOstMOdel. The original COCOMO model became one of the most widely used and discussed software cost estimation models in the industry. It has evolved intoa more comprehensive estimation model, called COCOMOII . Like its predecessor, COCOMO II is actually a hierarchy of estimation models that address the following areas:

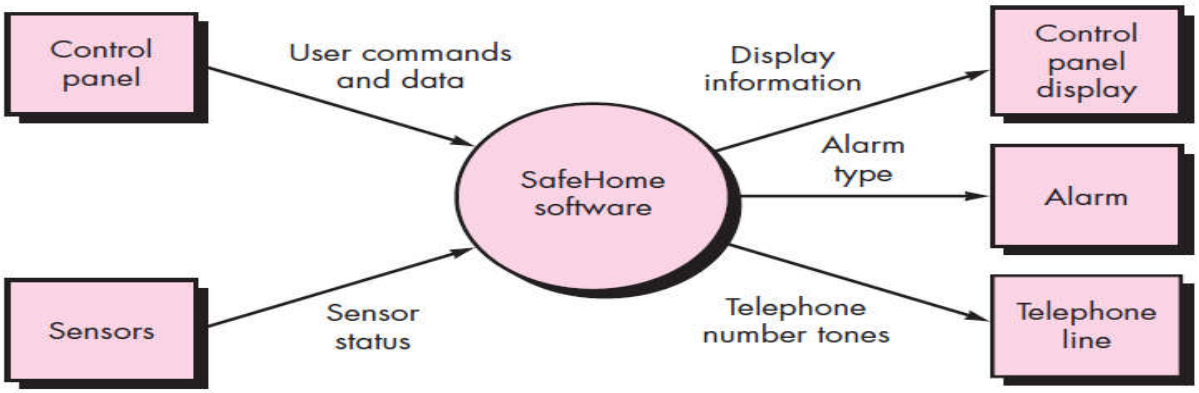
• Application composition model. Used during the early stages of software engineering, when prototyping of user interfaces, consideration of software and system interaction, assessment of performance, and evaluation of technology maturity are paramount.

• Early design stage model. Used once requirements have been stabilized and basic software architecture has been established.

• Post-architecture-stage model. Used during the construction of the software. Like all estimation models for software, the COCOMO II models require sizing information. Three different sizing options are available as part of the model hierarchy: object points, function points, and lines of source code.

Context model

The fundamental system modelor context diagram depicts the security function as a single transformation, representingthe external producers and consumers of data that flow into and out of thefunction. Figure below depicts a level 0 context model



Context-level DFD for the SafeHome security function

The behavioral model

The behavioral model indicates how software will respond to external events or stimuli. To create the model, you should perform the following steps:

1. Evaluate all use cases to fully understand the sequence of interaction within the system.
2. Identify events that drive the interaction sequence and understand how theseevents relate to specific objects.
3. Create a sequence for each use case.
4. Build a state diagram for the system.
5. Review the behavioral model to verify accuracy and consistency.

Identifying Events with the Use Case

In general, an event occurs whenever the system and an actor exchange information.

State Representations

In the context of behavioral modeling, two different characterizations of states must be considered: (1) the state of each class as the system performs its function and (2) the state of the system as observed from the outside as the system performs its function.

The state of a class takes on both passive and active characteristics.

A passive state is simply the current status of all of an object's attributes.

State diagrams for analysis classes.

One component of a behavioral model is a UML state diagram that represents active states for each class and the events (triggers) that cause changes between these active states. Figure below illustrates a state diagram for the Control Panel object in the SafeHome security function.

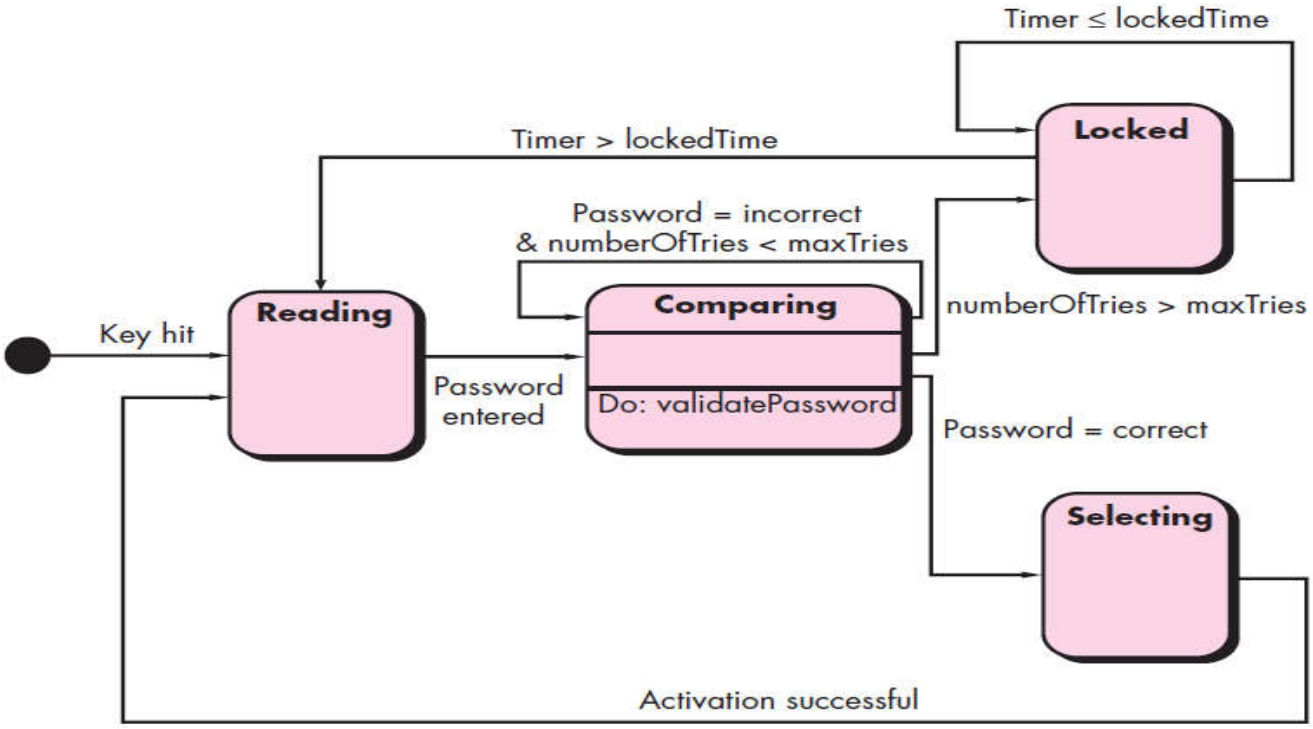


Figure : State diagram for the ControlPanel class

Sequence diagrams. The second type of behavioral representation, called a sequence diagram in UML, indicates how events cause transitions from object to object. Once events have been identified by examining a use case, the modeler creates a sequence diagram—a representation of how events cause flow from one object to another as a function of time. In essence, the sequence diagram is a shorthand version of the use case. It represents key classes and the events that cause behavior to flow from class to class.

DATA MODEL

If software requirements include the need to create, extend, or interface with a database or if complex data structures must be constructed and manipulated, the software team may choose to create a data model as part of overall requirements modeling. A software engineer or analyst defines all data objects that are processed within the system, the relationships between the data

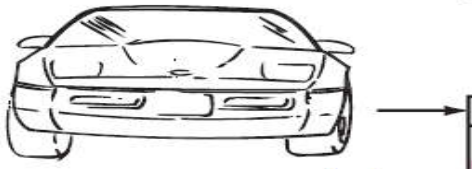
objects, and other information that is pertinent to the relationships. The entity-relationship diagram (ERD) addresses these issues and represents all data objects that are entered, stored, transformed, and produced within an application.

Data Objects

A data object is a representation of composite information that must be understood by software. By composite information, I mean something that has a number of different properties or attributes. Therefore, width (a single value) would not be a valid data object, but dimensions (incorporating height, width, and depth) could be defined as an object.

Data Attributes

Data attributes define the properties of a data object and take on one of three different characteristics. They can be used to (1) name an instance of the data object, (2) describe the instance, or (3) make reference to another instance in another table. In addition, one or more of the attributes must be defined as an identifier—that is, the identifier attribute becomes a "key" when we want to find an instance of the data object. In some cases, values for the identifier(s) are unique, although this is not a requirement. Referring to the data object car, a reasonable identifier might be the ID number.



Instance

Naming attributes

Ties one data object to another, in this case, owner

Identifier

Descriptive attributes

Referential attributes

Make	Model	ID#	Body type	Color	Owner
Lexus	LS400	AB123. . .	Sedan	White	RSP
Chevy	Corvette	X456. . .	Sports	Red	CCD
BMW	750iL	XZ765. . .	Coupe	White	LJL
Ford	Taurus	Q12A45. . .	Sedan	Blue	BLF

Relationships

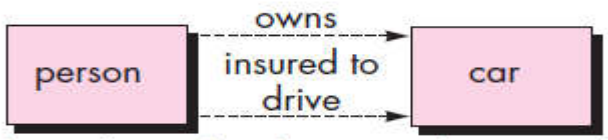
Data objects are connected to one another in different ways. Consider the two data objects, person and car. These objects can be represented using the simple notation illustrated in Figure below. A connection is established between person and car because the two objects are related. But what are the relationships? To determine the answer, you should understand the role of people (owners, in this case) and cars within the context of the software to be built. You can establish a set of object/relationship pairs that define the relevant relationships.

For example,

• A person owns a car.

• A person is insured to drive a car.

(a) A basic connection between data objects



(b) Relationships between data objects

Design Process

Software design is an iterative process through which requirements are translatedinto a blueprint" for constructing the software. Initially, the blueprint depicts a holisticview of software. That is, the design is represented at a high level of abstraction—a level that can be directly traced to the specific system objective and more detaileddata, functional, and behavioral requirements. As design iterations occur, subsequentrefinement leads to design representations at much lower levels of abstraction.These can still be traced to requirements, but the connection is more subtle.

Design Quality

Throughout the design process, the quality of the evolving design is assessed with a series of technical reviews. In order to evaluate the quality of a design representation, you and other members of the software team must establish technical criteria for good design.

Quality Guidelines

1. A design should exhibit an architecture that has been created using recognizable architectural styles or patterns is composed of componentsthat exhibit good design characteristics and can be implemented in an evolutionary fashion,2 thereby facilitating implementation and testing.
2. A design should be modular; that is, the software should be logically partitionedinto elements or subsystems.

3. A design should contain distinct representations of data, architecture, interfaces, and components.
4. A design should lead to data structures that are appropriate for the classes tobe implemented and are drawn from recognizable data patterns.
5. A design should lead to components that exhibit independent functionalcharacteristics.
6. A design should lead to interfaces that reduce the complexity of connectionsbetween components and with the external environment.
7. A design should be derived using a repeatable method that is driven by informationobtained during software requirements analysis.
8. A design should be represented using a notation that effectively communicatesits meaning.

Cohesion and coupling

Cohesionis an indication of the relative functional strength of a module. *Coupling* is an indication of the relative interdependence among modules. Cohesion is a natural extension of the information-hiding concept. A cohesive module performs a single task, requiring little interaction with other components in other parts of a program. Stated simply, a cohesive module should (ideally) do just one thing. Although you should always strive for high cohesion (i.e., single-mindedness), it is often necessary and advisable to have a software component perform multiple functions. However, "schizophrenic" components (modules that perform many unrelated functions) are to be avoided if a good design is to be achieved.

Coupling is an indication of interconnection among modules in a software structure.Coupling depends on the interface complexity between modules, the point atwhich entry or reference is made to a module, and what data pass across the interface.In software design, you should strive for the lowest possible coupling. Simpleconnectivity among modules results in software that is easier to understand and lessprone to a "ripple effect", caused when errors occur at one location and propagatethroughout a system.

Object Oriented Concepts

Requirements modeling (also called analysis modeling) focuses primarily on classes that are extracted directly from the statement of the problem. These entity classes typically represent things that are to be stored in a database and persist throughout the duration of the application

(unless they are specifically deleted). Design refines and extends the set of entity classes. Boundary and controller classes are developed and/or refined during design. Boundary classes create the interface (e.g., interactive screen and printed reports) that the user sees and interactswith, as the software is used. Boundary classes are designed with the responsibility of managing the way entity objects are represented to users.

Controller classes are designed to manage:

(1) the creation or update of entityobjects

(2) the instantiation of boundary objects as they obtain information from entity objects,

(3) complex communication between sets of objects, and

(4) validation of data communicated between objects or between the user and the application.

The concepts discussed in the paragraphs that follow can be useful in analysis and design work.

Inheritance.

Inheritance is one of the key differentiators between conventional and object-oriented systems. A subclass Y inherits all of the attributes and operations associated with its superclass X. This means that all data structures and algorithms originally designed and implemented for X are immediately available for Y—no further work need be done. Reuse has been accomplished directly. Any change to the attributes or operations contained within a superclass is immediately inherited by all subclasses. Therefore, the class hierarchy becomes a mechanism through which changes (at high levels) can be immediately propagated through a system. It is important to note that at each level of the class hierarchy new attributes and operations may be added to those that have been inherited from higher levels in the hierarchy. In fact, whenever a new class is to be created, you have a number of options:

• The class can be designed and built from scratch. That is, inheritance is not used.

• The class hierarchy can be searched to determine if a class higher in the hierarchy contains most of the required attributes and operations. The new class inherits from the higher class and additions may then be added, as required.

• The class hierarchy can be restructured so that the required attributes and operations can be inherited by the new class.

• Characteristics of an existing class can be overridden, and different versions of attributes or operations are implemented for the new class.

Like all fundamental design concepts, inheritance can provide significant benefit for the design, but if it is used inappropriately, it can complicate a design unnecessarily and lead to error-prone software that is difficult to maintain.

Messages.

Classes must interact with one another to achieve design goals. A message stimulates some behavior to occur in the receiving object. The behavior is accomplished when an operation is executed.

Polymorphism. *Polymorphism* is a characteristic that greatly reduces the effort required to extend the design of an existing object-oriented system. To understand polymorphism, consider a conventional application that must draw four different types of graphs: line graphs, pie charts, histograms, and Kiviat diagrams. Ideally, once data are collected for a particular type of graph, the graph should draw itself.To accomplish this in a conventional application (and maintain module cohesion), it would be necessary to develop drawing modules for each type of graph.

UML

The Unified Modeling Language (UML) is "a standard language for writingsoftware blueprints. UML may be used to visualize, specify, construct, anddocument the artifacts of a software-intensive system". In otherwords, just as building architects create blueprints to be used by a constructioncompany, software architects create UML diagrams to help software developersbuild the software.

To model classes, including their attributes, operations, and their relationships and associations with other classes, UML provides a class diagram. A class diagram provides a static or structural view of a system. It does not show the dynamic nature of the communications between the objects of the classes in the diagram. The main elements of a class diagram are boxes, which are the icons used to represent classes and interfaces. Each box is divided into horizontal parts. The top part contains the name of the class. The middle section lists the attributes of the class. An attribute refers to something that an object of that class knows or can provide all the time. Attributes are usually implemented as fields of the class, but they need not be. They could be values that the class can compute from its instance variables or values that the class can get from other objects of which it is composed.

For example, an object might always know the current time and be able to return it to you whenever you ask. Therefore, it would be appropriate to list the current time as an attribute of

that class of objects. However, the object would most likely not have that time stored in one of its instance variables, because it would need to continually update that field. Instead, the object would likely compute the current time (e.g., through consultation with objects of other classes) at the moment when the time is requested. The third section of the class diagram contains the operations or behaviors of the class. An operation refers to what objects of the class can do. It is usually implemented as a method of the class.

Figure A1 presents a simple example of a Thoroughbred class that models thoroughbred horses. It has three attributes displayed—mother, father, and birthyear. The diagram also shows three operations: getCurrentAge(), getFather(), and getMother(). There may be other suppressed attributes and operations not shown in the diagram.

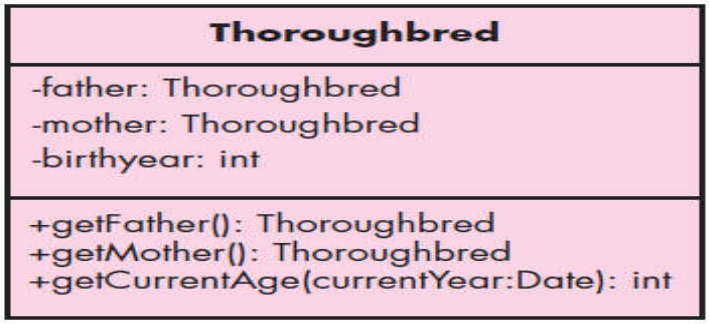


Figure A1 : A class diagram for a Thoroughbred class

Each attribute can have a name, a type, and a level of visibility. The type and visibility are optional. The type follows the name and is separated from the name by a colon. The visibility is indicated by a preceding –, #, ~, or +, indicating, respectively, private, protected, package, or public visibility. In Figure A1, all attributes have private visibility, as indicated by the leading minus sign (–). You can also specify that an attribute is a static or class attribute by underlining it. Each operation can also be displayed with a level of visibility, parameters with names and types, and a return type. An abstract class or abstract method is indicated by the use of italics for the name in the class diagram. See the Horse class in Figure A2 for an example. An interface is indicated by adding the phrase "«interface»" (called a stereotype) above the name. See the OwnedObject interface in Figure A2. An interface can also be represented graphically by a hollow circle. It is worth mentioning that the icon representing a class can have other optional parts. For example, a fourth section at the bottom of the class box can be used to list the responsibilities of the class. This fourth section is not shown in any of the figures in this appendix. Class diagrams can also show relationships between classes. A class that is a subclass

of another class is connected to it by an arrow with a solid line for its shaft and with a triangular hollow arrowhead. The arrow points from the subclass to the superclass. In UML, such a relationship is called a generalization. For example, in Figure A2, the Thoroughbred and QuarterHorse classes are shown to be subclasses of the Horse abstract class. An arrow with a dashed line for the arrow shaft indicates implementation of an interface. In UML, such a relationship is called a realization. For example, in Figure A2, the Horse class implements or realizes the OwnedObject interface.

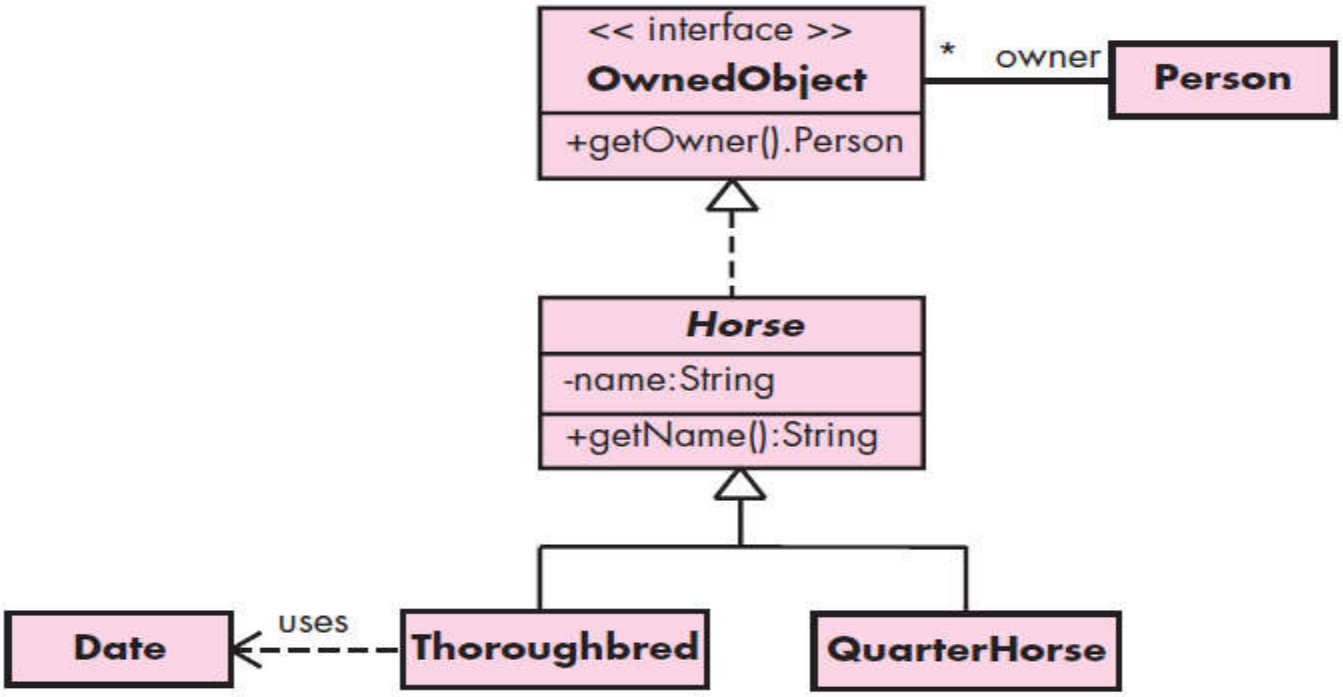


Figure A2: A class diagram regarding horses

An ***association*** between two classes means that there is a structural relationship between them. Associations are represented by solid lines. An association has many optional parts. It can be labeled, as can each of its ends, to indicate the role of each class in the association. For example, in Figure A2, there is an association between **OwnedObject**and **Person** in which the **Person** plays the role of owner. Arrows on either or both ends of an association line indicate navigability. Also, each end of the association line can have a multiplicity value displayed. Navigability and multiplicity are explained in more detail later in this section. An association might also connect a class with itself, using a loop. Such an association indicates the connection

of an object of the class with other objects of the same class. An association with an arrow at one end indicates one-way navigability. The arrow means that from one class you can easily access the second associated class to which the association points, but from the second class, you cannot necessarily easily access the first class. Another way to think about this is that the firs class is aware of the second class, but the second class object is not necessarily directly aware of the first class. An association with no arrows usually indicates a two-way association, which is what was intended in Figure A2, but it could also just mean that the navigability is not important and so was left off.

It should be noted that an attribute of a class is very much the same thing as an association of the class with the class type of the attribute. That is, to indicate that a class has a property called "name" of type String, one could display that property as an attribute, as in the **Horse** class in Figure A2. Alternatively, one could create a one-way association from the **Horse** class to the **String** class with the role of the **String** class being "name." The attribute approach is better for primitive data types, whereas the association approach is often better if the property's class plays a major role in the design, in which case it is valuable to have a class box for that type. A *dependency* relationship represents another connection between classes and is indicated by a dashed line (with optional arrows at the ends and with optional labels). One class depends on another if changes to the second class might require changes to the first class. An association from one class to another automatically indicates a dependency. No dashed line is needed between classes if there is already an association between them.

However, for a transient relationship (i.e., a class that does not maintain any long-term connection to another class but does use that class occasionally) we should draw a dashed line from the first class to the second. For example, in Figure A2, the **Thoroughbred** class uses the **Date** class whenever its *getCurrentAge()* method is invoked, and so the dependency is labeled "uses." The *multiplicity* of one end of an association means the number of objects of that class associated with the other class. A multiplicity is specified by a nonnegative integer or by a range of integers. A multiplicity specified by "0..1" means that there are 0 or 1 objects on that end of the association. For example, each person in the world has either a Social Security number or no such number (especially if they are not U.S. citizens), and so a multiplicity of 0..1 could be used in an association between a**Person** class and a **SocialSecurityNumber**class in a class diagram. A multiplicityspecified by "1..*" means one or more, and a multiplicity specified by "0..*" or just

“*“means zero or more. An * was used as the multiplicity on the **OwnedObject**end ofthe association with class **Person** in Figure A2 because a **Person** can own zero ormore objects..

If one end of an association has multiplicity greater than 1, then the objects of the class referred to at that end of the association are probably stored in a collection, such as a set or ordered list. One could also include that collection class itself in the UML diagram, but such a class is usually left out and is implicitly assumed to be there due to the multiplicity of the association.

An***aggregation***is a special kind of association indicated by a hollow diamond on one end of the icon. It indicates a “whole/part” relationship, in that the class to which the arrow points is considered a “part” of the class at the diamond end of the association.

A ***composition***is an aggregation indicating strong ownership of the parts. In a composition, the parts live and die with the owner because they have no role in the software system independent of the owner.

UML *use-case diagram* help you determine the functionality and features of the software from the user’s perspective. To give you a feeling for how use cases and use-case diagrams work, we will create some for a software application for managing digital music files, similar to Apple’s iTunes software.

Some of the things the software might do include:

• Download an MP3 music file and store it in the application’s library.

• Capture streaming music and store it in the application’s library.

• Manage the application’s library (e.g., delete songs or organize them in playlists).

• Burn a list of the songs in the library onto a CD.

• Load a list of the songs in the library onto an iPod or MP3 player.

• Convert a song from MP3 format to AAC format and vice versa.

A use-case diagram for the digital music application is shown in Figure A3.

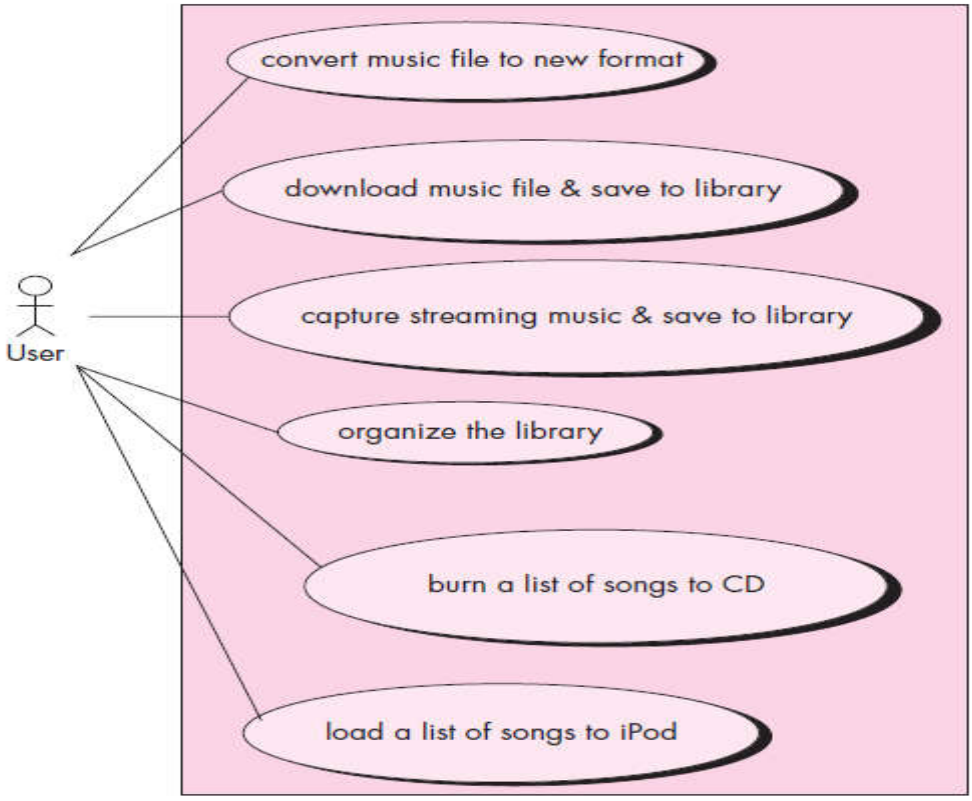


Figure A3: Use case diagram for music system

RISK ANALYSIS

Software development is activity that uses a variety of technological advancements and requires high levels of knowledge. Because of these and other factors, every software development project contains elements of uncertainty. This is known as project risk. The success of a software development project depends quite heavily on the amount of risk that corresponds to each project activity. As a project manager, it's not enough to merely be aware of the risks. To achieve a successful outcome, project leadership must identify, assess, prioritize, and manage all of the major risks. Risk is the possibility of suffering loss, and total risk exposure to a specific project will account for both the *probability* and the *size* of the potential loss.

Risk management

Risk management means risk containment and mitigation. First, you've got to identify and plan. Then be ready to act when a risk arises, drawing upon the experience and knowledge of the entire team to minimize the impact to the project. Risk management includes the following tasks:

- ***Identify*** risks and their triggers
- ***Classify*** and prioritize all risks
- Craft a ***plan*** that links each risk to a mitigation
- ***Monitor*** for risk triggers during the project
- Implement the ***mitigating action*** if any risk materializes
- ***Communicate*** risk status throughout project

Identify and Classify Risks

Most software engineering projects are inherently risky because of the variety potential problems that might arise. Experience from other software engineering projects can help managers classify risk. The importance here is not the elegance or range of classification, but rather to precisely identify and describe all of the real threats to project success. A simple but effective classification scheme is to arrange risks according to the areas of impact.

Five Types of Risk In Software Project Management

For most software development projects, we can define five main risk impact areas:

- New, unproven technologies
- User and functional requirements
- Application and system architecture
- Performance
- Organizational

New, unproven technologies. The majority of software projects entail the use of new technologies. Ever-changing tools, techniques, protocols, standards, and development systems increase the probability that technology risks will arise in virtually any substantial software engineering effort. Training and knowledge are of critical importance, and the improper use of new technology most often leads directly to project failure.

User and functional requirements. Software requirements capture all user needs with respect to the software system features, functions, and quality of service. Too often, the process of requirements definition is lengthy, tedious, and complex. Moreover, requirements usually change with discovery, prototyping, and integration activities. Change in elemental requirements will likely propagate throughout the entire project, and modifications to user requirements might not translate to functional requirements. These disruptions often lead to one or more critical failures of a poorly-planned software development project.

Application and system architecture. Taking the wrong direction with a platform, component, or architecture can have disastrous consequences. As with the technological risks, it is vital that the team includes experts who understand the architecture and have the capability to make sound design choices.

Performance. It's important to ensure that any risk management plan encompasses user and partner expectations on performance. Consideration must be given to benchmarks and threshold testing throughout the project to ensure that the work products are moving in the right direction.

Organizational. Organizational problems may have adverse effects on project outcomes. Project management must plan for efficient execution of the project, and find a balance between the needs of the development team and the expectations of the customers. Of course, adequate staffing includes choosing team members with skill sets that are a good match with the project.

Risk Management Plan

After cataloging all of the risks according to type, the software development project manager should craft a risk management plan. As part of a larger, comprehensive project plan, the risk management plan outlines the response that will be taken for each risk—if it materializes.

Monitor and Mitigate

To be effective, software risk monitoring has to be integral with most project activities. Essentially, this means frequent checking during project meetings and critical events.

Monitoring includes:

- Publish project status reports and include risk management issues
- Revise risk plans according to any major changes in project schedule
- Review and reprioritize risks, eliminating those with lowest probability
- Brainstorm on potentially new risks after changes to project schedule or scope

When a risk occurs, the corresponding mitigation response should be taken from the risk management plan.

Mitigating options include:

- **Accept:** Acknowledge that a risk is impacting the project. Make an explicit decision to accept the risk without any changes to the project. Project management approval is mandatory here.
- **Avoid:** Adjust project scope, schedule, or constraints to minimize the effects of the risk.

- **Control:** Take action to minimize the impact or reduce the intensification of the risk.
- **Transfer:** Implement an organizational shift in accountability, responsibility, or authority to other stakeholders that will accept the risk.
- **Continue Monitoring:** Often suitable for low-impact risks, monitor the project environment for potentially increasing impact of the risk.

Communicate

Throughout the project, it's vital to ensure effective communication among all stakeholders, managers, developers, QA—especially marketing and customer representatives. Sharing information and getting feedback about risks will greatly increase the probability of project success.

UNIT-III

SYLLABUS

KARPAGAM ACADEMY OF HIGHER EDUCATION

CLASS:IIM.Com(CA)

COURSE NAME: SOFTWARE MODELS AND ENGINEERING

COURSECODE:18CCP301

BATCH-2018-2020

S/W Requirements, S/W Metrices& Testing Strategies: S/W Requirements : Functional and non-functional requirements, User requirements, System requirements.SRA& SRS. S/W Metrices: Process Metrices, Project Metrices& Product Metrices. Testing Strategies : A strategic approach to software testing, Testing fundamentals, Test Case Design. Types Of Testing: Black-Box Testing, White-Box Testing, Validation testing, System testing, the art of Debugging. Code walkthrough and reviews. Software Quality, Metrics for Analysis Model, Metrics for Design Model, Metrics for source code, Metrics for testing, Metrics for maintenance.

Requirements engineering (RE) is the process of establishing the services that the customer requires from a system and the constraints under which it operates and is developed. The **requirements** themselves are the descriptions of the system services and constraints that are generated during the requirements engineering process. Requirements may range from a high-level abstract statement of a service or of a system constraint to a detailed mathematical functional specification. As much as possible, requirements should describe **what** the system should do, but **not how** it should do it.

A functional requirement describes *what* **a software system should do, while non-functional requirements place constraints on** *how* **the system will do so.**

In software engineering, a functional requirement defines a system or its component. It describes the functions a software must perform. A function is nothing but inputs, its behavior, and outputs. It can be a calculation, data manipulation, business process, user interaction, or any other specific functionality which defines what function a system is likely to perform.

Functional software requirements help you to capture the intended behavior of the system. This

behavior may be expressed as functions, services or tasks or which system is required to perform.

Functional requirements

The functional requirements for a system describe what the system should do. These requirements depend on the type of software being developed, the expected users of the software, and the general approach taken by the organization when writing requirements. When expressed as user requirements, functional requirements are usually described in an abstract way that can be understood by system users. However, more specific functional system requirements describe the system functions, its inputs and outputs, exceptions, etc., in detail. Functional system requirements vary from general requirements covering what the system should do to very specific requirements reflecting local ways of working or an organization's existing systems.

For example, here are examples of functional requirements for the MHC-PMS system, used to maintain information about patients receiving treatment for mental health problems:

1. A user shall be able to search the appointments lists for all clinics.

2. The system shall generate each day, for each clinic, a list of patients who are expected to attend appointments that day.

3. Each staff member using the system shall be uniquely identified by his or her eight-digit employee number.

These functional user requirements define specific facilities to be provided by the system. These have been taken from the user requirements document and they show that functional requirements may be written at different levels of detail (contrast requirements 1 and 3). Imprecision in the requirements specification is the cause of many software engineering problems. It is natural for a system developer to interpret an ambiguous requirement in a way that simplifies its implementation. Often, however, this is not what the customer wants. New requirements have to be established and changes made to the system. Of course, this delays system delivery and increases costs. For example, the first example requirement for the MHC-

PMS states that a user shall be able to search the appointments lists for all clinics. The rationale for this requirement is that patients with mental health problems are sometimes confused. They may have an appointment at one clinic but actually go to a different clinic. If they have an appointment, they will be recorded as having attended, irrespective of the clinic. The medical staff member specifying this may expect 'search' to mean that, given a patient name, the system looks for that name in all appointments at all clinics. However, this is not explicit in the requirement. System developers may interpret the requirement in a different way and may implement a search so that the user has to choose a clinic then carry out the search. This obviously will involve more user input and so take longer. In principle, the functional requirements specification of a system should be both complete and consistent. Completeness means that all services required by the user should be defined. Consistency means that requirements should not have contradictory definitions. In practice, for large, complex systems, it is practically impossible to achieve requirements consistency and completeness. One reason for this is that it is easy to make mistakes and omissions when writing specifications for complex systems. Another reason is that there are many stakeholders in a large system. A stakeholder is a person or role that is affected by the system in some way. Stakeholders have different— and often inconsistent—needs. These inconsistencies may not be obvious when the requirements are first specified, so inconsistent requirements are included in the specification. The problems may only emerge after deeper analysis or after the system has been delivered to the customer.

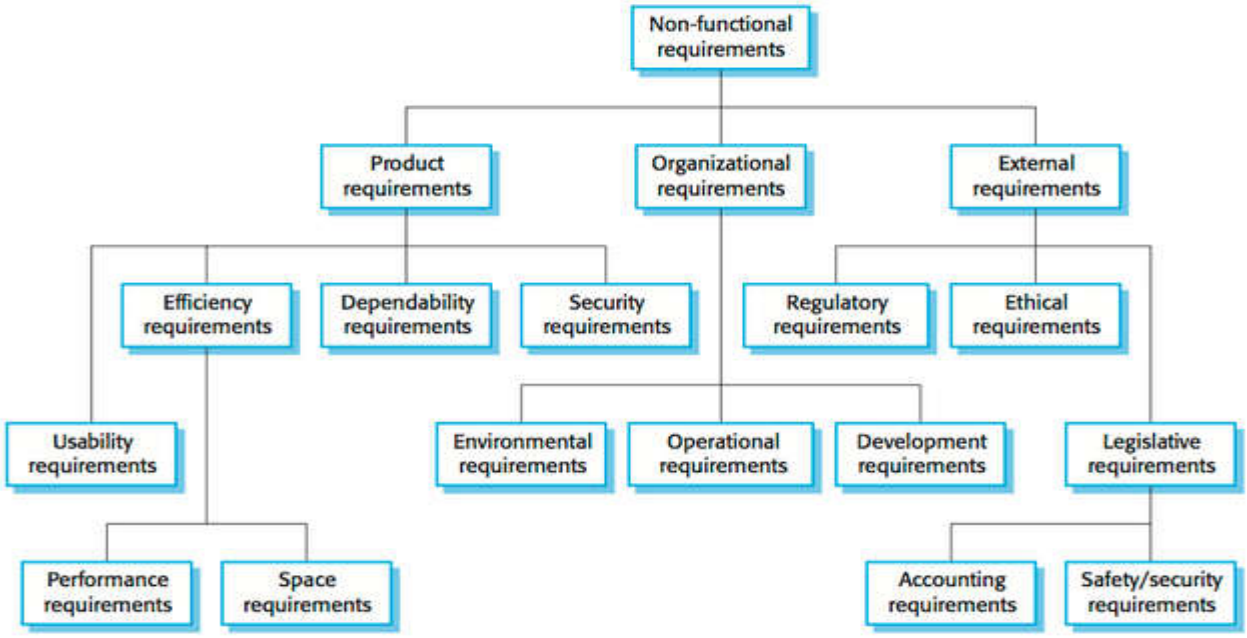
Non Functional Requirements

Non-functional requirements, as the name suggests, are requirements that are not directly concerned with the specific services delivered by the system to its users. They may relate to emergent system properties such as reliability, response time, and store occupancy. Alternatively, they may define constraints on the system implementation such as the capabilities of I/O devices or the data representations used in interfaces with other systems. Non-functional requirements, such as performance, security, or availability, usually specify or constrain characteristics of the system as a whole. Non-functional requirements are often more critical than individual functional requirements. System users can usually find ways to work around a system function that doesn't really meet their needs. However, failing to meet a non-functional requirement can mean that the whole system is unusable. For example, if an aircraft system does

not meet its reliability requirements, it will not be certified as safe for operation; if an embedded control system fails to meet its performance requirements, the control functions will not operate correctly. Although it is often possible to identify which system components implement specific functional requirements (e.g., there may be formatting components that implement reporting requirements), it is often more difficult to relate components to non-functional requirements. The implementation of these requirements may be diffused throughout the system. There are two reasons for this:

1. Non-functional requirements may affect the overall architecture of a system rather than the individual components. For example, to ensure that performance requirements are met, you may have to organize the system to minimize communications between components.

2. A single non-functional requirement, such as a security requirement, may generate a number of related functional requirements that define new system services that are required. In addition, it may also generate requirements that restrict existing requirements.



Types of non-functional requirement

Three classes of non-functional requirements:

1. Product requirements

Requirements which specify that the delivered product must behave in a particular way e.g. execution speed, reliability, etc.

2. Organizational requirements

Requirements which are a consequence of organizational policies and procedures e.g. process standards used, implementation requirements, etc.

3. External requirements

Requirements which arise from factors which are external to the system and its development process e.g. interoperability requirements, legislative requirements, etc.

Non-functional requirements may be very difficult to state precisely and imprecise requirements may be difficult to verify. If they are stated as a **goal** (a general intention of the user such as ease of use), they should be rewritten as a **verifiable** non-functional requirement (a statement using some quantifiable metric that can be objectively tested). Goals are helpful to developers as they convey the intentions of the system users.

User requirements

High-level abstract requirements written as statements, in a natural language plus diagrams, of what services the system is expected to provide to system users and the constraints under which it must operate. The **user requirement(s)** document (URD) or user requirement(s) specification (URS) is a document usually used in software engineering that specifies what the user expects the software to be able to do. Once the required information is completely gathered it is documented in a URD, which is meant to spell out exactly what the software must do and becomes part of the contractual agreement. A customer cannot demand features not in the URD, whilst the developer cannot claim the product is ready if it does not meet an item of the URD. The URD can be used as a guide to planning cost, timetables, milestones, testing, etc. The explicit nature of the URD allows customers to show it to various stakeholders to make sure all necessary features are described. Formulating a URD requires negotiation to determine what is technically and economically feasible. Preparing a URD is one of those skills that lies between a science and an art, requiring both software technical skills and interpersonal skills.

The user requirements for a system should describe the functional and nonfunctional requirements so that they are understandable by system users who don't have detailed technical knowledge. Ideally, they should specify only the external behavior of the system. The requirements document should not include details of the system architecture or design. Consequently, if you are writing user requirements, you should not use software jargon, structured notations, or formal notations. You should write user requirements in natural language, with simple tables, forms, and intuitive diagrams.

User requirements are almost always written in natural language supplemented by appropriate diagrams and tables in the requirements document. System requirements may also be written in natural language but other notations based on forms, graphical system models, or mathematical system models can also be used. Graphical models are most useful when you need to show how a state changes or when you need to describe a sequence of actions. UML sequence charts and state charts show the sequence of actions that occur in response to a certain message or event. Formal mathematical specifications are sometimes used to describe the requirements for safety- or security-critical systems, but are rarely used in other circumstances.

System requirements

System requirements are expanded versions of the user requirements that are used by software engineers as the starting point for the system design. They add detail and explain how the user requirements should be provided by the system. They may be used as part of the contract for the implementation of the system and should therefore be a complete and detailed specification of the whole system. Ideally, the system requirements should simply describe the external behavior of the system and its operational constraints. They should not be concerned with how the system should be designed or implemented. However, at the level of detail required to completely specify a complex software system, it is practically impossible to exclude all design information. There are several reasons for this:

1. You may have to design an initial architecture of the system to help structure the requirements specification. The system requirements are organized according to the different sub-systems that make up the system.
2. In most cases, systems must interoperate with existing systems, which constrain the design and impose requirements on the new system.

3. The use of a specific architecture to satisfy non-functional requirements may be necessary. An external regulator who needs to certify that the system is safe may specify that an already certified architectural design be used.

Software requirements specification (SRS)

It is a document that describes what the software will do and how it will be expected to perform. An SRS describes the functionality the product needs to fulfill all stakeholders (business, users) needs. A software requirements specification (SRS) is a document that captures complete description about how the system is expected to perform. It is usually signed off at the end of requirements engineering phase.

A software requirements specification (SRS) is a document that is created when a detailed description of all aspects of the software to be built must be specified before the project is to commence. It is important to note that a formal SRS is not always written. In fact, there are many instances in which effort expended on an SRS might be better spent in other software engineering activities. However, when software is to be developed by a third party, when a lack of specification would create severe business issues, or when a system is extremely complex or business critical, an SRS may be justified.

Quality Characteristics of a good SRS

Correctness:

User review is used to ensure the correctness of requirements stated in the SRS. SRS is said to be correct if it covers all the requirements that are actually expected from the system.

Completeness:

Completeness of SRS indicates every sense of completion including the numbering of all the pages, resolving the to be determined parts to as much extent as possible as well as covering all the functional and non-functional requirements properly.

Consistency:

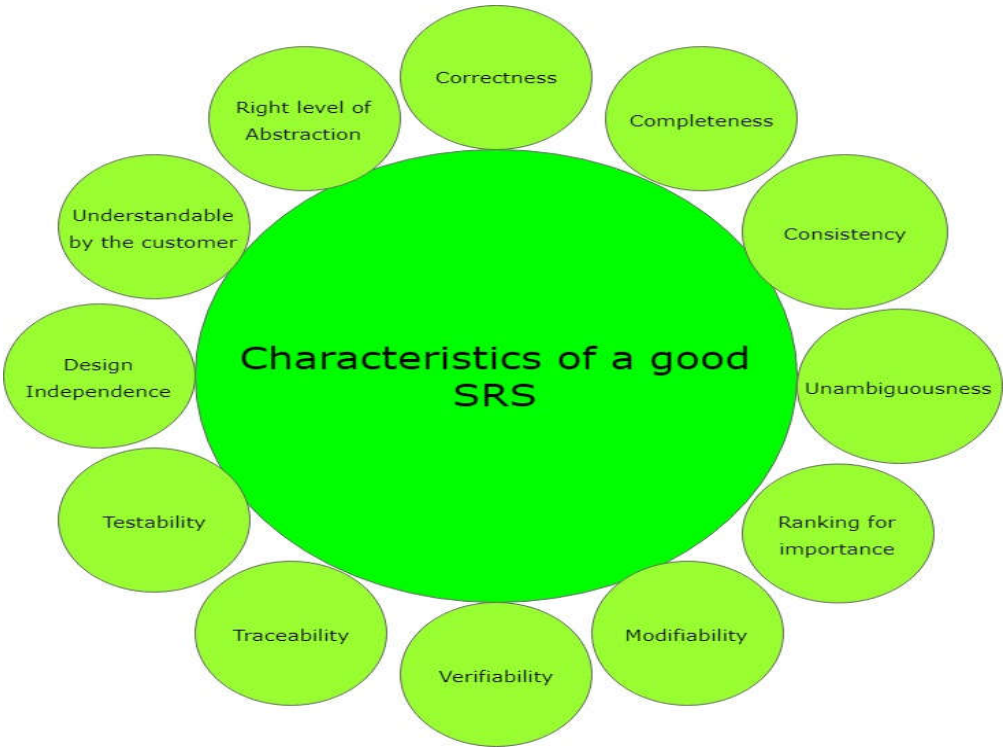
Requirements in SRS are said to be consistent if there are no conflicts between any set of

requirements. Examples of conflict include differences in terminologies used at separate places, logical conflicts like time period of report generation, etc.

 Unambiguousness:
An SRS is said to be unambiguous if all the requirements stated have only 1 interpretation. Some of the ways to prevent unambiguousness include the use of modelling techniques like ER diagrams, proper reviews and buddy checks, etc.

 Ranking for importance and stability:
There should a criterion to classify the requirements as less or more important or more specifically as desirable or essential. An identifier mark can be used with every requirement to indicate its rank or stability.

 Modifiability:
SRS should be made as modifiable as possible and should be capable of easily accepting changes to the system to some extent. Modifications should be properly indexed and cross-referenced.



 Verifiability:
An SRS is verifiable if there exists a specific technique to quantifiably measure the extent to which every requirement is met by the system. For example, a requirement stating that the system must be user-friendly is not verifiable and listing such requirements should be avoided.

 Traceability:
One should be able to trace a requirement to a design component and then to a code segment in the program. Similarly, one should be able to trace a requirement to the corresponding test cases.

 Design Independence:
There should be an option to choose from multiple design alternatives for the final system. More specifically, the SRS should not include any implementation details.

 Testability:
An SRS should be written in such a way that it is easy to generate test cases and test plans from the document.

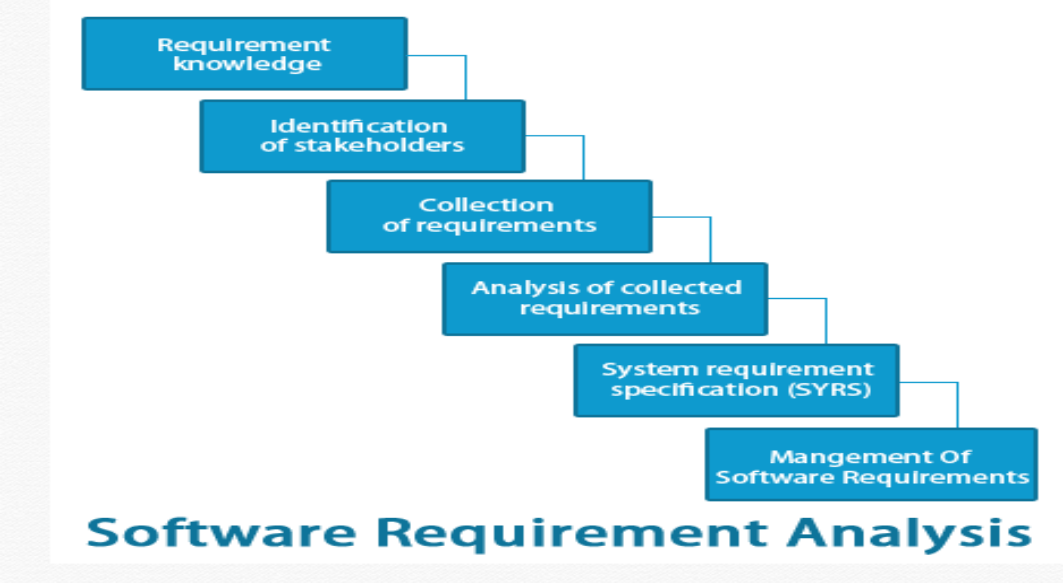
 Understandable by the customer:
An end user maybe an expert in his/her specific domain but might not be an expert in computer science. Hence, the use of formal notations and symbols should be avoided to as much extent as possible. The language should be kept easy and clear.

 Right level of abstraction:
If the SRS is written for the requirements phase, the details should be explained explicitly. Whereas, for a feasibility study, fewer details can be used. Hence, the level of abstraction varies according to the purpose of the

Software Requirement Analysis(SRA)

Software requirement is a functional or non-functional need to be implemented in the system. Functional means providing particular service to the user. For example, in context to banking application the functional requirement will be when customer selects "View Balance" they must be able to look at their latest account balance. Software requirement can also be a non-functional,

it can be a performance requirement. For example, a non-functional requirement is where every page of the system should be visible to the users within 5 seconds.



Necessity Of Requirement Analysis

According to statistics major reason of failure of software is that it does not meet with the requirement of the user. Requirement analysis involves the task that determines the needs of the software, which mainly includes complaints and needs of various clients/stakeholders.

Software Requirement Analysis Process

The steps for effective capturing on present requirements of users are:

- Requirement Knowledge:

It is very necessary to know about the requirements of the users before starting any project. Working on the present requirements of the users will be helpful in gaining popularity of your project.

- Identification of Stakeholders:

Stakeholders includes customers, end-users, system administrators etc. identifying the correct stakeholder is second step and is one of the most important step in all. Identifying the correct stakeholders help to properly analyze and create a road map for gathering requirements.

- Collection of Requirements:

After identifying stakeholders one has to collect requirements for them. Based on the nature and aim of the project there can be many kinds of stakeholders. Interacting with stakeholder groups can be in person interviews, focus groups, market study, surveys and secondary research.

- Analysis of Collected Requirements:

Once the data is gathered structured analysis must be done of the data to make models. Data are analysed on the basis of various parameters depending on the goals of the software. These include animation, automated reasoning, knowledge based critiquing, consistency checking, analogical and case based reasoning.

- System requirement Specification (SYRS):

Once the data is analyzed they are put together in the form of system requirement specification document (SYRS) or system requirement specification (SRS). It acts as a blueprint for the designing team to make the project. It serves as a technical collection of all the requirements of stake holders which includes user requirements, system requirements, user interface and operational requirements.

- Management Of Software Requirements:

The last step of this analysis process is correcting and validating all elements of requirement specifications document. Errors can be corrected at this stage. Minor changes can also be done according to the requirement of the software user.

Code Walkthrough is a form of peer review in which a programmer leads the review process and the other team members ask questions and spot possible errors against development standards and other issues.

- The meeting is usually led by the author of the document under review and attended by other members of the team.
- Review sessions may be formal or informal.
- Before the walkthrough meeting, the preparation by reviewers and then a review report with a list of findings.
- The scribe, who is not the author, marks the minutes of meeting and note down all the defects/issues so that it can be tracked to closure.
- The main purpose of walkthrough is to enable learning about the content of the document under review to help team members gain an understanding of the content of the document and also to find defects.

KARPAGAM ACADEMY OF HIGHER EDUCATION

CLASS:IIM.Com(CA)

COURSE NAME: SOFTWARE MODELS AND ENGINEERING

COURSECODE:18CCP301

BATCH-2018-2020

UNIT-I

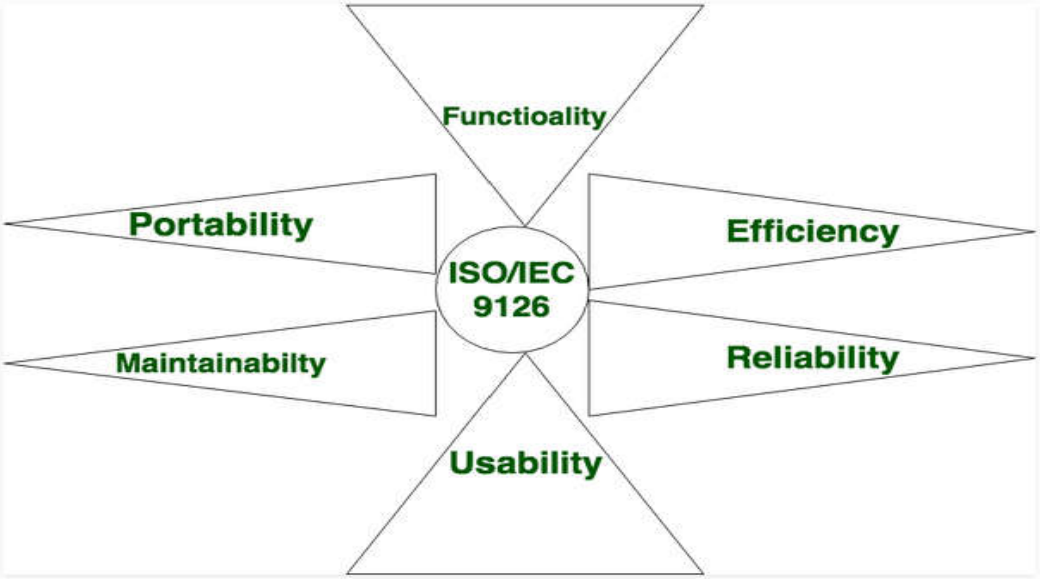
SYLLABUS

Fundamentals of Software Engineering and Process models : Definition, Software characteristics and Application. Software myths, Software engineering- A layered technology and SDLC. Software process models: Linear sequential model, prototyping model, RAD Model. Evolutionary process models: Incremental process models and Spiral model. Component based ,4GT. Maturity Models: CMM, CMMI, PCMM, PSP, TSP, Process patterns, process assessment. Unified process: SEI CMM and ISO 9001. PSP and Six Sigma. Clean room technique.

Software Engineering definition proposed by Fritz Bauer at the seminal conference on the subject still serves as a basis for discussion:

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

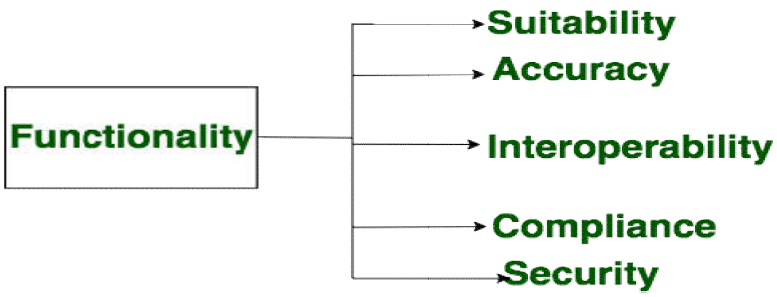
Software is defined as collection of computer programs, procedures, rules and data. Software Characteristics are classified into six major components:



These components are described below:

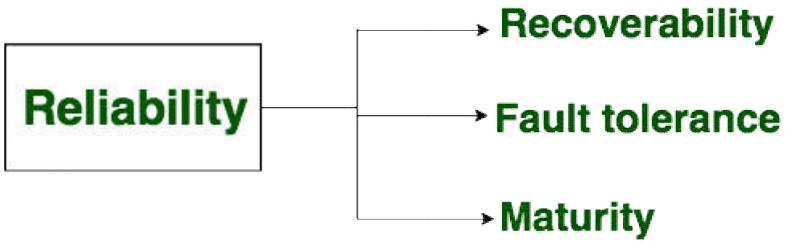
Functionality:

It refers to the degree of performance of the software against its intended purpose. Required functions are:



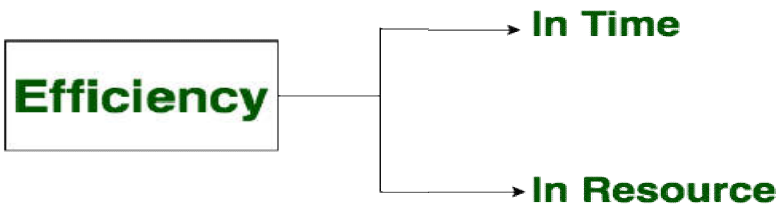
Reliability:

A set of attribute that bear on capability of software to maintain its level of performance under the given condition for a stated period of time. Required functions are:



- **Efficiency:**

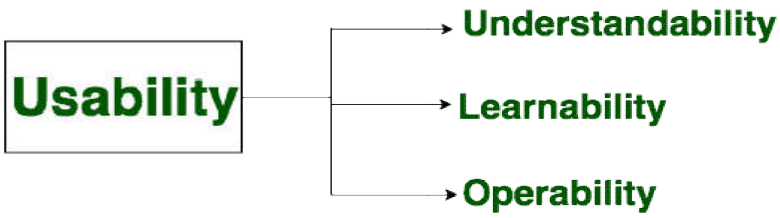
  It refers to the ability of the software to use system resources in the most effective and efficient manner. The software should make effective use of storage space and executive command as per desired timing requirement. Required functions are:



- **Usability:**

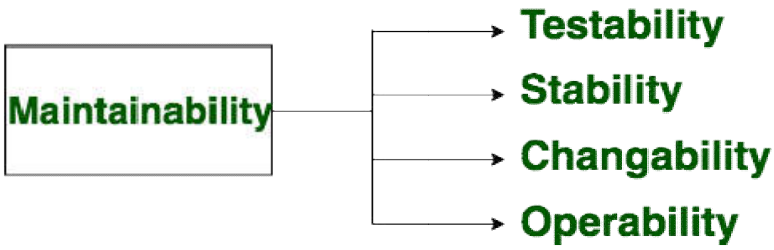
  It refers to the extent to which the software can be used with ease, the amount of effort or time required to learn how to use the software.

  Required functions are:



- **Maintainability:**

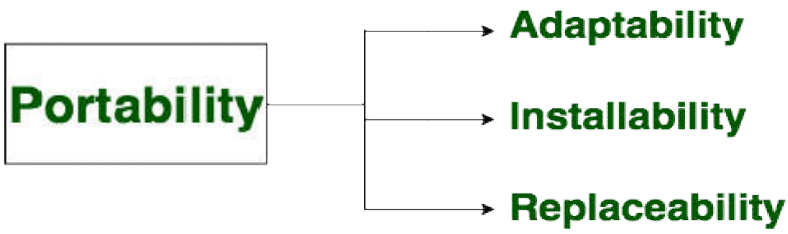
  It refers to the ease with which the modifications can be made in a software system to extend its functionality, improve its performance, or correct errors. Required functions are:



- **Portability:**

  A set of attribute that bear on the ability of software to be transferred from one environment to another, without or minimum changes.

  Required functions are:



Software myths—It is erroneous belief about software and the process that is used to build it—can be traced to the earliest days of computing. Myths have a number of attributes that make them insidious. For instance, they appear to be reasonablestatements of fact (sometimes containing elements of truth), they have an intuitive feel, and they are often promulgated by experienced practitioners who "know the score." Today, most knowledgeable software engineering professionals recognize myths for what they are—misleading attitudes that have caused serious problems for managers and practitioners alike. However, old attitudes and habits are difficult to modify, and remnants of software myths remain.

Management myths. Managers with software responsibility, like managers inmost disciplines, are often under pressure to maintain budgets, keep schedules fromslipping, and improve quality. Like a drowning person who grasps at a straw, a softwaremanager often grasps at belief in a software myth, if that belief will lessen thepressure (even temporarily).

Myth: We already have a book that's full of standards and procedures forbuilding software. Won't that provide my people with everything theyneed to know?Reality: Are softwarepractitioners aware of its existence? Does it reflect modernsoftware engineering practice? Is it complete? Is itstreamlined to improve time-to-delivery while still maintaining afocus on quality? In many cases, the answer to all of these questionsis "no."

Myth: If we get behind schedule, we can add more programmers and catch up(sometimes called the "Mongolian horde" concept).

Reality: Software development is not a mechanistic process like manufacturing.In the words of Brooks : "adding people to a late softwareproject makes it later." At first, this statement may seemcounterintuitive. However, as new people are added, people whowere working must spend time educating the newcomers, therebyreducing the amount of time spent on productive developmenteffort. People can be added but only in a planned and coordinatedmanner.

Myth: If I decide to outsource the software project to a third party, I can justrelax and let that firm build it.

Reality: If an organization does not understand how to manage and controlsoftware projects internally, it will invariably struggle when it outsourcessoftware projects.

Customer myths. A customer who requests computer software may be a personat the next desk, a technical group

down the hall, the marketing/sales department,or an outside company that has requested software under contract. In many cases,the customer believes myths about software because software managers and practitionersdo little to correct misinformation. Myths lead to false expectations (by thecustomer) and, ultimately, dissatisfaction with the developer.

Myth: A general statement of objectives is sufficient to begin writingprograms—we can fill in the details later.
Reality: Although a comprehensive and stable statement of requirements isnot always possible, an ambiguous "statement of objectives" is arecipe for disaster. Unambiguous requirements (usually derivediteratively) are developed only through effective and continuouscommunication between customer and developer.

Myth: Software requirements continually change, but change can be easilyaccommodated because software is flexible.
Reality: It is true that software requirements change, but the impact ofchange varies with the time at which it is introduced. When requirementschanges are requested early (before design or code has beenstarted), the cost impact is relatively small. However, as timepasses, the cost impact grows rapidly—resources have been committed,a design framework has been established, and change cancause upheaval that requires additional resources and major designmodification.

Practitioner's myths. Myths that are still believed by software practitioners havebeen fostered by over 50 years of programming culture. During the early days, programmingwas viewed as an art form. Old ways and attitudes die hard.

Myth: Once we write the program and get it to work, our job is done.Reality: Someone once said that "the sooner you begin 'writing code,' thelonger it'll take you to get done." Industry data indicate that between60 and 80 percent of all effort expended on software will be expendedafter it is delivered to the customer for the first time.

Myth: Until I get the program "running" I have no way of assessing its quality.
Reality: One of the most effective software quality assurance mechanismscan be applied from the inception of a project—the technical review.

Myth: The only deliverable work product for a successful project is the workingprogram.

Reality: A working program is only one part of a software configuration thatincludes many elements. A variety of work products (e.g., models, documents, plans) provide a foundation for successful engineeringand, more important, guidance for software support.

Myth: Software engineering will make us create voluminous and unnecessarydocumentation and will invariably slow us down.

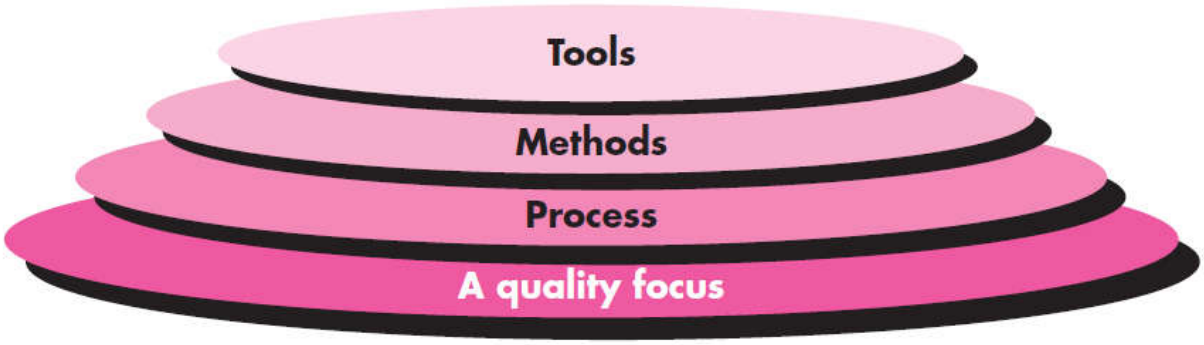
Reality: Software engineering is not about creating documents. It is aboutcreating a quality product. Better quality leads to reduced rework.And reduced rework results in faster delivery times.

Layered Technology

Software engineering is a layered technology. Referring to Figure 1.3, any engineering approach (including software engineering) must rest on an organizational commitment to quality. Total quality management, Six Sigma, and similar philosophies foster a continuous process improvement culture, and it is this culture that ultimately leads to the development of increasingly more effective approaches to software engineering. The bedrock that supports software engineering is a quality focus. The foundation for software engineering is the process layer. The software engineeringprocess is the glue that holds the technology layers together and enables rational and timely development of computer software.

FIGURE 1.3

Software engineering layers



Software engineering is Divided into **4** layers:-

1. A quality Process :-

- Any engineering approach must rest on quality.
- The "Bed Rock" that supports software Engineering is Quality Focus.

2. Process :-

- Foundation for SE is the Process Layer
- SE process is the GLUE that holds all the technology layers together and enables the timely development of computer software.

- It forms the base for management control of software project.

3. Methods :-

- SE methods provide the "Technical Questions" for building Software.
- Methods contain a broad array of tasks that include communication requirement analysis, design modeling, program construction testing and support.

4. Tools :-

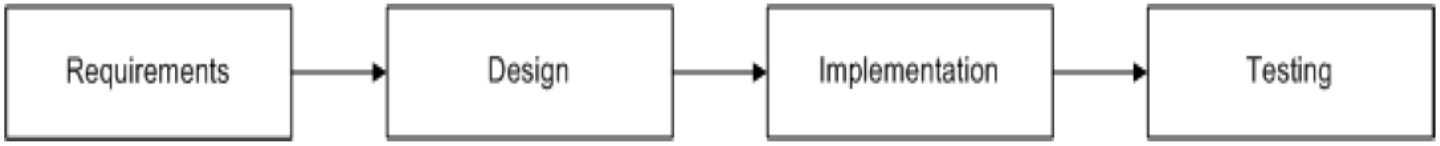
- SE tools provide automated or semi-automated support for the "Process" and the "Methods".
- Tools are integrated so that information created by one tool can be used by another.

A maturity level is a well-defined evolutionary plateau toward achieving a mature software process. Each maturity level provides a layer in the foundation for continuous process improvement.

Software Life Cycle

A Life Cycle shows how a living thing borns, grows, lives, and dies. The stages from birth to death.Software life cycle model is the stages of development that a software development goes through. The following figure shows the stages of software development.

Software life cycle models describe phases of the software cycle and the order in which those phases are executed. There are tons of models, and many companies adopt their own, but all have very similar patterns. The general, basic model is shown below:



Each phase produces deliverables required by the next phase in the life cycle. Requirements are translated into design. Code is produced during implementation that is driven by the design. Testing verifies the deliverable of the implementation phase against requirements.

A **Software Process** can be defined as set of activities, methods, practices and transformations which people employ to develop and maintain software and the associated products. The quality of a software product is essentially determined by the quality of the processes employed to develop and maintain it.

The Linear Sequential Model

This is a software process model that involves a systematic progression through **analysis**, **design**, **coding**, **testing** and **maintenance** phases. It is also referred to as the "**waterfall model**".

Also known as the classic life cycle or waterfall model, it suggests a systematic, sequential approach to software development. Problems with this approach are:

• Real projects rarely follow the sequential flow and changes can cause confusion.

• This model has difficulty accommodating requirements change

• The customer will not see a working version until the project is nearly complete

• Developers are often blocked unnecessarily, due to previous tasks not being done

The Prototyping Model

Advantages:

- Easy and quick to identify customer requirements

- Customers can validate the prototype at the earlier stage and provide their inputs and feedback

- Good to deal with the following cases:

1. Customer cannot provide the detailed requirements
2. Very complicated system-user interactions
3. Use new technologies, hardware and algorithms
4. Develop new domain application systems

Problems:

-The prototype can serve as **—the first system**.

-Developers usually attempt to develop the product based on the prototype.

-Developers often make implementation compromises in order to get a prototyping that is working quickly.

-Customers may be unaware that the prototype is not a product, which is held with.

The RAD Model

Rapid Application Development (RAD) is a linear sequential software development process model that emphasizes an extremely short development cycle.

- A —high-speed‖ adaptation of linear sequential model

- Component-based construction

- Effective when requirements are well understood and project scope is constrained.

Advantages:

- Short development time

- Cost reduction due to software reuse and component-based construction

Problems:

- For large, but scalable projects, RAD requires sufficient resources.

- RAD requires developers and customers who are committed to the schedule.
- Constructed software is project-specific, and may not be well modularized.
- Its quality depends on the quality of existing components.
- Not appropriate projects with high technical risk and new technologies.

Incremental Process Models

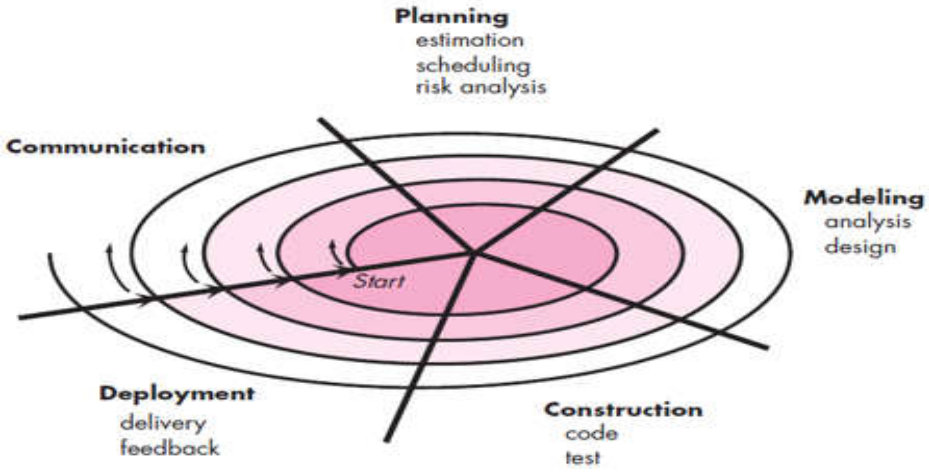
There are many situations in which initial software requirements are reasonably well defined, but the overall scope of the development effort precludes a purely linear process. In addition, there may be a compelling need to provide a limited set of software functionality to users quickly and then refine and expand on that functionalityin later software releases. In such cases, you can choose a process model that is designed to produce the software in increments. The incremental model combines elements of linear and parallel process flows

Incremental model applies linear sequences in a staggered fashion as calendar time progresses. Each linear sequence produces deliverable "increments" of the software in a manner that is similar to the increments produced by an evolutionary process flow.For example, word-processing software developed using the incremental paradigm might deliver basic file management, editing, and document production functionsin the first increment; more sophisticated editing and document production capabilities in the second increment; spelling and grammar checking in the third increment; and advanced page layout capability in the fourth increment. It should be noted that the process flow for any increment can incorporate the prototyping paradigm.

When an incremental model is used, the first increment is often a core product. That is, basic requirements are addressed but many supplementary features (some known, others unknown) remain undelivered. The core product is used by the customer (or undergoes detailed evaluation).

The Spiral Model.

Originally proposed by Barry Boehm, the spiral model is an evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the waterfall model. It providesthe potential for rapid development of increasingly more complete versions of the software. Boehm describes the model in the following manner: The spiral development model is a risk-driven process model generator that is used to guide multi-stakeholder concurrent engineering of software intensive systems. It has two main distinguishing features. One is a cyclic approach for incrementally growing a system's degree of definition and implementation while decreasing its degree of risk. The other is a set of anchor point milestones for ensuring stakeholder commitment to 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 model or prototype. During later iterations, increasingly more complete versions of the engineered system are produced.



A Typical Spiral Model

Component-based development model

Commercial off-the-shelf (COTS) software components, developed by vendors who offer them as products, provide targeted functionality with well-defined interfaces that enable the component to be integrated into the software that is to be built. The component-based development model incorporates many of the characteristics of the spiral model. It is evolutionary in nature, demanding an iterative approach to the creation of software. However, the component-based development model constructs applications from prepackaged software components. Modeling and construction activities begin with the identification of candidate components. These components can be designed as either conventional software modules or object-oriented classes or packages of classes. Regardless of the technology that is used to create the components, the component-based development model incorporates the following steps (implemented using an evolutionary approach):

1. Available component-based products are researched and evaluated for theapplication domain in question.
2. Component integration issues are considered.
3. A software architecture is designed to accommodate the components.
4. Components are integrated into the architecture.
5. Comprehensive testing is conducted to ensure proper functionality.

The component-based development model leads to software reuse, and reusability provides software engineers with a number of measurable benefits. Your software engineering team can achieve a reduction in development cycle time as well as a reduction in project cost if component reuse becomes part of your culture.

4GT Process Model

4GT begins with from “Requirement Gathering” this process go through the customer, the customer go illustrate the requirements. The customer could actually describe the requirements and these would be directly translated

into an operational prototype. If the product is a smaller product this process may be possible to move directly from requirements gathering step to implementation using a non-procedural fourth generation language (4GL), for lager products this procedure may little hard, therefor it's necessary to use the design strategy in 4GT.When it comes to large projects, the design phase it is crucial to avoid poor quality, poor maintainability. To transform a 4GL implementation into a product, the developer must conduct through testing, develop meaningful documentation, and perform all other solution integration activities. The 4GT developed software must be built in a manner that enables maintenance to be performed expeditiously. There are some merits to summarize the current features of 4GT approach. In the 4GL implementation the code can be generated based on some specification. The 4GT developed software must be built in a manner that enables maintenance to be performed expeditiously. There are some merits to summarize the current features of 4GT approach. The use of 4GT is a viable approach for many different application areas coupled with computer- aided software engineering tools and code generators, 4GT offers a credible solution to many software problem.

Benefits of the 4GT

Flexible: The Fourth Generation applications are Modifiable by Design, which means they are designed from the beginning to accommodate change. They are easily modifiable, either by you, the customer, or by your Fourth Generation Authorized reseller to your specifications.

Scalable: The Fourth Generation applications are Modifiable by Design, which means they are designed from the beginning to accommodate change.They are easily modifiable, either by you, the customer, or by your Fourth Generation Authorized reseller to your specifications.

Total Data Access: Your data represents your company's greatest single asset. The worth of that asset, however, is directly related to your ability to record it and access it.

The Fourth Generation Technique (4GT) is based on NPL that is the Non-Procedural Language techniques. Depending upon the specifications made the 4GT move towards uses various tools for the automatic generation of source codes. It is the very important tool which make use of the non-procedural language for Report generation, Database query, Manipulation of data, Interaction of screen, Definition, Generation of code, Spread Sheet capabilities, and High level graphical capacity etc. 4GT begins with a requirement-gathering stage. The customer would illustrate requirements and these would be directly converted into an unworkable operational prototype. For small applications, it may be possible to move directly from requirements gathering step to implementation using a non-procedural fourth generation language (4GL), however for large application it is necessary to develop a design strategy for the system even if a 4GL is to be used. Implementation using a 4GT enables the software developer to represent desired result in a manner that leads to automatic generation of code to create those results, obviously,

data structure with relevant information must exist and be readily accessible by the 4GL. To transform a 4GL implementation into a product, the developer must conduct through testing, develop meaningful documentation, and perform all other solution integration activities. The 4GT developed software must be built in a manner that enables maintenance to be performed expeditiously. There are some merits to summarize the current features of 4GT approach. The use of 4GT is a viable approach for many different application areas coupled with computer-aided software engineering tools and code generators, 4GT offers a credible solution to many software problem. Data collected from companies that use 4Gt indicates that the time required to produce software is greatly reduced for small and intermediate application is also reduced. However the use of 4GT for large software development efforts demands as much or more analysis design and testing to achieve substantial timesaving that result from the elimination of coding.

Compatibility Maturity Model -CMM

Maturity level 1 _Initial

organizations often produce products and services that work; however, they frequently exceed the budget and schedule of their projects.Maturity level 1 organizations are characterized by a tendency to over commit, abandon processes in the time of crisis, and not be able to repeat their past successes.

Maturity Level 2 - Managed

At maturity level 2, an organization has achieved all the specific and generic goals of the maturity level 2 process areas. In other words, the projects of the organization have ensured that requirements are managed and that processes are planned, performed, measured, and controlled.The process discipline reflected by maturity level 2 helps to ensure that existing practices are retained during times of stress. When these practices are in place, projects are performed and managed according to their documented plans.At maturity level 2, requirements, processes, work products, and services are managed. The status of the work products and the delivery of services are visible to management at defined points.Commitments are established among relevant stakeholders and are revised as needed. Work products are reviewed with stakeholders and are controlled.The work products and services satisfy their specified requirements, standards, and objectives.

Maturity Level 3 - Defined

At maturity level 3, an organization has achieved all the specific and generic goals of the process areas assigned to maturity levels 2 and 3.At maturity level 3, processes are well characterized and understood, and are described in standards, procedures, tools, and methods.A critical distinction between maturity level 2 and maturity level 3 is the scope of standards, process descriptions, and procedures. At maturity level 2, the standards, process descriptions, and procedures may be quite different in each specific instance of the process (for example, on a particular project). At maturity level 3, the standards, process descriptions, and procedures for a project are tailored from the

organization's set of standard processes to suit a particular project or organizational unit. The organization's set of standard processes includes the processes addressed at maturity level 2 and maturity level 3. As a result, the processes that are performed across the organization are consistent except for the differences allowed by the tailoring guidelines.Another critical distinction is that at maturity level 3, processes are typically described in more detail and more rigorously than at maturity level 2. At maturity level 3, processes are managed more proactively using an understanding of the interrelationships of the process activities and detailed measures of the process, its work products, and its services.

Maturity Level 4 - Quantitatively managed

At maturity level 4, an organization has achieved all the specific goals of the process areas assigned to maturity levels 2, 3, and 4 and the generic goals assigned to maturity levels 2 and 3.At maturity level 4 Sub processes are selected that significantly contribute to overall process performance. These selected sub processes are controlled using statistical and other quantitative techniques.Quantitative objectives for quality and process performance are established and used as criteria in managing processes. Quantitative objectives are based on the needs of the customer, end users, organization, and process implementers. Quality and process performance are understood in statistical terms and are managed throughout the life of the processes.

Maturity Level 5 - Optimizing

At maturity level 5, an organization has achieved all the specific goals of the process areas assigned to maturity levels 2, 3, 4, and 5 and the generic goals assigned to maturity levels 2 and 3.Processes are continually improved based on a quantitative understanding of the common causes of variation inherent in processes.Maturity level 5 focuses on continually improving process performance through both incremental and innovative technological improvements.Quantitative process-improvement objectives for the organization are established, continually revised to reflect changing business objectives, and used as criteria in managing process improvement.

Capability Maturity Model Integration - CMMI

Capability Maturity Model Integration (CMMI) is a process level improvement training and appraisal program. Administered by the CMMI Institute, a subsidiary of ISACA, it was developed at Carnegie Mellon University (CMU). It is required by many United States Department of Defense (DoD) and U.S. Government contracts, especially in software development. CMU claims CMMI can be used to guide process improvement across a project, division, or an entire organization. CMMI defines the following maturity levels for processes: Initial, Repeatable, Defined, Quantitatively Managed, and Optimizing.

CMMI Model

1) Initial: The software process is characterized as ad hoc, and occasionally even chaotic. Few processes are defined, and success depends on individual effort and heroics.

2) Repeatable: Basic project management processes are established to track cost, schedule, and functionality. The necessary process discipline is in place to repeat earlier successes on projects with similar applications.

3) Defined: The software process for both management and engineering activities is documented, standardized, and integrated into a standard software process for the organization. All projects use an approved, tailored version of the organization's standard software process for developing and maintaining software.

4) Managed: Detailed measures of the software process and product quality are collected. Both the software process and products are quantitatively understood and controlled.

5) Optimizing: Continuous process improvement is enabled by quantitative feedback from the process and from piloting innovative ideas and technologies.

People Capability Maturity Model (PCMM)

People Capability Maturity Model (PCMM) is a maturity framework that focuses on continuously improving the management and development of the human assets of a software or information systems organization. PCMM can be perceived as the application of the principles of Capability Maturity Model to human assets of a software organization. It describes an evolutionary improvement path from ad hoc, inconsistently performed practices, to a mature, disciplined, and continuously improving development of the knowledge, skills, and motivation of the workforce. Although the focus in People CMM is on software or information system organizations, the processes

and practices are applicable for any organization that aims to improve the capability of its workforce. PCMM will be guiding and effective particularly for organizations whose core processes are knowledge intensive.The primary objective of the People Capability Maturity Model is to improve the capability of the entire workforce. This can be defined as the level of knowledge, skills, and process abilities available for performing an organization's current and future business activities.

10 Principles of People Capability Maturity Model (PCMM)

The People Capability Maturity Model describes an evolutionary improvement path from ad hoc, inconsistently performed workforce practices, to a mature infrastructure of practices for continuously elevating workforce capability. The philosophy implicit the PCMM can be summarized in ten principles.In mature organizations, workforce capability is directly related to business performance.
Workforce capability is a competitive issue and a source of strategic advantage.
Workforce capability must be defined in relation to the organization's strategic business objectives.
Knowledge-intense work shifts the focus from job elements to workforce competencies.
Capability can be measured and improved at multiple levels, including individuals, workgroups, workforce competencies, and the organization.
An organization should invest in improving the capability of those workforce competencies that are critical to its core competency as a business.
Operational management is responsible for the capability of the workforce.
The improvement of workforce capability can be pursued as a process composed from proven practices and procedures.
The organization is responsible for providing improvement opportunities, while individuals are responsible for taking advantage of them.
Since technologies and organizational forms evolve rapidly, organizations must continually evolve their workforce practices and develop new workforce competencies.
The People Capability Maturity Model (People CMM) is a roadmap for implementing workforce practices that continuously improve the capability of an organization's workforce. Since an organization cannot implement all of the best workforce practices in an afternoon, the People CMM introduces them in stages. Each progressive level of the People CMM produces a unique transformation in the organization's culture by equipping it with more powerful practices for attracting, developing, organizing, motivating, and retaining its workforce. Thus, the People CMM establishes an integrated system of workforce practices that matures through increasing alignment with the organization's business objectives, performance, and changing needs.

Although the People CMM has been designed primarily for application in knowledge intense organizations, with appropriate tailoring it can be applied in almost any organizational setting. The People CMM's primary objective is to improve the capability of the workforce. Workforce capability can be defined as the level of knowledge, skills, and process abilities available for performing an organization's business activities.

Personal Software Process (PSP)

Every developer uses some process to build computer software. The process may be haphazard or ad hoc; may change on a daily basis; may not be efficient, effective, or even successful; but a "process" does exist. Watts Humphrey [Hum97] suggests that in order to change an ineffective personal process, an individual must move through four phases, each requiring training and careful instrumentation. The Personal Software Process (PSP) emphasizes personal measurement of both the work product that is produced and the resultant quality of the work product. In addition PSP makes the practitioner responsible for project planning (e.g., estimating and scheduling) and empowers the practitioner to control the quality of all software work products that are developed. The PSP model defines five framework activities:

Planning.

This activity isolates requirements and develops both size and resource estimates. In addition, a defect estimate (the number of defects projected for the work) is made. All metrics are recorded on worksheets or templates. Finally, development tasks are identified and a project schedule is created.

High-level design. External specifications for each component to be constructed are developed and a component design is created. Prototypes are built when uncertainty exists. All issues are recorded and tracked.

High-level design review. Formal verification methods are applied to uncover errors in the design. Metrics are maintained for all important tasks and work results.

Development. The component-level design is refined and reviewed. Code is generated, reviewed, compiled, and tested. Metrics are maintained for all important tasks and work results.

Postmortem. Using the measures and metrics collected (this is a substantial amount of data that should be analyzed statistically), the effectiveness of the process is determined. Measures and metrics should provide guidance formodifying the process to improve its effectiveness.

Team Software Process (TSP)

Because many industry-grade software projects are addressed by a team of practitioners, Watts Humphrey extended the lessons learned from the introduction of PSP and proposed a Team Software Process (TSP). The goal of TSP is to build a "selfdirected" project team that organizes itself to produce high-quality software.

Humphrey defines the following objectives for TSP:

• Build self-directed teams that plan and track their work, establish goals, andown their processes and plans. These

can be pure software teams or integratedproduct teams (IPTs) of 3 to about 20 engineers.

• Show managers how to coach and motivate their teams and how to helpthem sustain peak performance.

• Accelerate software process improvement by making Level 5behavior normal and expected.

• Provide improvement guidance to high-maturity organizations.

• Facilitate university teaching of industrial-grade team skills.

A self-directed team has a consistent understanding of its overall goals and objectives; defines roles and responsibilities for each team member; tracks quantitativeproject data (about productivity and quality); identifies a team process that is appropriate for the project and a strategy for implementing the process; defines local standards that are applicable to the team's software engineering work; continually assesses risk and reacts to it; and tracks, manages, and reports project status.

TSP defines the following framework activities: project launch, high-level design, implementation, integration and test, and postmortem. Like their counterparts in PSP (note that terminology is somewhat different), these activities enable the team to plan, design, and construct software in a disciplined manner while at the same time quantitatively measuring the process and the product. The postmortem sets the stage for process improvements.

PROCESS PATTERNS

Every software team encounters problems as it moves through the software process. It would be useful if proven solutions to these problems were readily available to the team so that the problems could be addressed and resolved quickly. A *process pattern* describes a process-related problem that is encountered during software engineering work, identifies the environment in which the problem has been encountered, and suggests one or more proven solutions to the problem. Stated in more general terms, a process pattern provides you with a template—a consistent method for describing problem solutions within the context of the software process. By combining patterns, a software team can solve problems and construct a process that best meets the needs of a project.

Patterns can be defined at any level of abstraction. In some cases, a pattern might be used to describe a problem (and solution) associated with a complete processmodel (e.g., prototyping). In other situations, patterns can be used to describe a problem (and solution) associated with a framework activity (e.g., **planning**) or an action within a framework activity (e.g., project estimating).

Ambler has proposed a template for describing a process pattern:

Pattern Name. The pattern is given a meaningful name describing it within the context of the software process (e.g., **TechnicalReviews**).

Forces. The environment in which the pattern is encountered and the issues that make the problem visible and may affect its solution.

Type. Ambler suggests three types of patterns:

1. Stage pattern—defines a problem associated with a framework activity for the process. Since a framework activity encompasses multiple actions andwork tasks, a stage pattern incorporates multiple task patterns (see the following) that are relevant to the stage (framework activity). An example of astage pattern might be EstablishingCommunication. This pattern would incorporate the task pattern RequirementsGathering and others.

2. Task pattern—defines a problem associated with a software engineering action or work task and relevant to successful software engineeringpractice (e.g., RequirementsGathering is a task pattern).

3. Phase pattern—define the sequence of framework activities that occurs within the process, even when the overall flow of activities is iterativein nature. An example of a phase pattern might be **SpiralModel** or **Prototyping.**

PROCESS ASSESSMENT

The existence of a software process is no guarantee that software will be delivered on time, that it will meet the customer's needs, or that it will exhibit the technical characteristics that will lead to long-term quality characteristics. Process patterns must be coupled with solid software engineering practice

(Part 2 of this book). In addition, the process itself can be assessed to ensure that it meets a set of basic process criteria that have been shown to be essential for a successful software engineering.. A number of different approaches to software process assessment and improvement have been proposed over the past few decades:

Standard CMMI Assessment Method for Process Improvement

(SCAMPI)—provides a five-step process assessment model that incorporates five phases: initiating, diagnosing, establishing, acting, and learning. The

SCAMPI method uses the SEI CMMI as the basis for assessment.

CMM-Based Appraisal for Internal Process Improvement (CBA IPI) — provides a diagnostic technique for assessing the relative maturity of asoftware organization; uses the SEI CMM as the basis for the assessment.

SPICE (ISO/IEC15504)—a standard that defines a set of requirements for software process assessment. The intent of the standard is to assist organizations

in developing an objective evaluation of the efficacy of any defined software process .

ISO 9001:2000 for Software—a generic standard that applies to any organization that wants to improve the overall quality of the products, systems,

or services that it provides. Therefore, the standard is directly applicable to software organizations and companies

Six Sigma for Software Engineering

Six Sigma is the most widely used strategy for statistical quality assurance in industry today. Originally popularized by Motorola in the 1980s, the Six Sigma strategy "is a rigorous and disciplined methodology that uses data and statistical analysis to measure and improve a company's operational performance by identifying and

eliminating defects' in manufacturing and service-related processes". The term Six Sigma is derived from six standard deviations—3.4 instances (defects) per million occurrences—implying an extremely high quality standard. The Six Sigma methodology defines three core steps:

- *Define* customer requirements and deliverables and project goals via welldefined methods of customer communication.
- *Measure* the existing process and its output to determine current quality performance (collect defect metrics).
- *Analyze* defect metrics and determine the vital few causes.

If an existing software process is in place, but improvement is required, Six Sigmasuggests two additional steps:

- *Improve* the process by eliminating the root causes of defects.
- *Control* the process to ensure that future work does not reintroduce the causes of defects.

Clean room technique (clean room design)

The clean room technique is a process in which a new product is developed by **reverse engineering** an existing product, and then the new product is designed in such a way that patent or **copyright** infringement is avoided. The clean room technique is also known as clean room design. (Sometimes the words "clean room" are merged into the single word, "cleanroom.") Sometimes this process is called the **Chinese wall** method, because the intent is to place a demonstrable intellectual barrier between the reverse engineering process and the development of the new product.

The use of the clean room technique can be compared, in some respects, with the fair use of copyrighted publications in order to compile a new document. For example, a new book about Linux can be authored on the basis of information obtained by researching existing books, articles, white papers, and Web sites. This does not necessarily constitute copyright infringement, even though other books on Linux already exist, and even if the new book contains essentially the same information as the existing publications. However, this is the case only as long as passages from the existing works are not copied verbatim or nearly verbatim, and as long as the new work does not have substantially the same structure as any of the existing works.

Use of the clean room technique puts engineers and enterprises in a legal gray area. If the owner of the original copyright or patent can demonstrate that the development of a new product was done by means of reverse engineering and is not significantly different from the existing product, a lawsuit may result. Any attempt to reverse engineer an existing product, and then create a new product based on the results of the reverse engineering process, should be undertaken only with the advice of a reputable attorney who is experienced in copyright infringement and reverse engineering issues**.**

KARPAGAM ACADEMY OF HIGHER EDUCATION

CLASS:IIM.Com(CA)

COURSE NAME: SOFTWARE MODELS AND ENGINEERING

COURSECODE:18CCP301

BATCH-2018-2020

Managing Software Projects & Design Engineering: The management spectrum, software quality, measurement and metrics. Software project estimation, decomposition techniques. Empirical estimation models (COCOMO), the Make & Buy Decision. System models: Context Models, Behavioral models, Data models, Object models. Design process, Design quality and design model. Fundamental issues in software design: Goodness of design, cohesions, coupling. Function-oriented design and object – oriented concepts. Architectural styles and patterns, Architectural Design: Unified Modeling Language (UML), User interface design. Risk Analysis and management.

MANAGEMENT SPECTRUM

Effective software project management focuses on the **four P's: people, product, process, and project**. The order is not arbitrary. The manager who forgets that software engineering work is an intensely human endeavor will never have success in project management. A manager who fails to encourage comprehensive stakeholder communication early in the evolution of a product risks building an elegant solution for the wrong problem. The manager who pays little attention to the process runs the risk of inserting competent technical methods and tools into a vacuum. The manager who embarks without a solid project plan jeopardizes the success of the project.

ThePeople

The cultivation of motivated, highly skilled software people has been discussed since the 1960s. In fact, the "people factor" is so important that the Software Engineering Institute has developed a *People Capability Maturity Model* (People-CMM), in recognition of the fact that "every organization needs to continually improve its ability to attract, develop, motivate, organize, and retain the workforce needed to accomplish its strategic business objectives". The people capability maturity model defines the following key practice areas for software people:

staffing, communication and coordination, work environment, performance management, training, compensation, competency analysis and development, career development, workgroup development, team/culture development, and others. Organizations that achieve high levels of People-CMM maturity have a higher likelihood of implementing effective software project management practices. The People-CMM is a companion to the *Software Capability Maturity Model– Integration* that guides organizations in the creation of a mature software process.

The Product

Before a project can be planned, product objectives and scope should be established, alternative solutions should be considered, and technical and management constraints should be identified. Without this information, it is impossible to define reasonable (and accurate) estimates of the cost, an effective assessment of risk, a realistic breakdown of project tasks, or a manageable project schedule that provides a meaningful indication of progress. As a software developer, you and other stakeholders must meet to define product objectives and scope. In many cases, this activity begins as part of the system engineering or business process engineering and continues as the first step in software requirements engineering. Objectives identify the overall goals for the product (from the stakeholders' points of view) without considering how these goals will be achieved. Scope identifies the primary data, functions, and behaviors that characterize the product, and more important, attempts to bound these characteristics in a quantitative manner. Once the product objectives and scope are understood, alternative solutions are considered. Although very little detail is discussed, the alternatives enable managers and practitioners to select a "best" approach, given the constraints imposed by delivery deadlines, budgetary restrictions, personnel availability, technical interfaces, and myriad other factors.

The Process

A software process provides the framework from which a comprehensive plan for software development can be established. A small number of framework activities are applicable to all software projects, regardless of their size or complexity. A number of different task sets—tasks, milestones, work products, and quality assurance points—enable the framework activities to be adapted to the characteristics of the software project and the requirements of the project team. Finally, umbrella activities—such as software quality assurance, software configuration management, and measurement—overlay the process model. Umbrella activities are independent of any one framework activity and occur throughout the process.

The Project

We conduct planned and controlled software projects for one primary reason—it is the only known way to manage complexity. And yet, software teams still struggle. In a study of 250 large software projects between 1998 and 2004, Capers Jones found that "about 25 were deemed successful in that they achieved their schedule, cost, and quality objectives. About 50 had delays or overruns below 35 percent, while about 175 experienced major delays

and overruns, or were terminated without completion." Although the success rate for present-day software projects may have improved somewhat, our project failure rate remains much higher than it should be. To avoid project failure, a software project manager and the software engineers who build the product must avoid a set of common warning signs, understand the critical success factors that lead to good project management, and develop a commonsense approach for planning, monitoring, and controlling the project.

Software quality, measurement and metrics

Software quality assurance is composed of a variety of tasks associated with two different constituencies—the software engineers who do technical work and an SQA group that has responsibility for quality assurance planning, oversight, record keeping, analysis, and reporting.

Software engineers address quality (and perform quality control activities) by applying solid technical methods and measures, conducting technical reviews, and performing well-planned software testing.

Measurements in the physical world can be categorized in two ways: direct measures (e.g., the length of a bolt) and indirect measures (e.g., the "quality" of bolts produced, measured by counting rejects). Software metrics can be categorized similarly.

Direct measures of the software process include cost and effort applied. Direct measures of the product include lines of code (LOC) produced, execution speed, memory size, and defects reported over some set period of time. Indirect measures of the product include functionality, quality, complexity, efficiency, reliability, maintainability, and many other "–abilities".

The cost and effort required to build software, the number of lines of code produced, and other direct measures are relatively easy to collect, as long as specific conventions for measurement are established in advance.

However, the quality and functionality of software or its efficiency or maintainability are more difficult to assess and can be measured only indirectly. The software metrics domain can be partitioned into process, project, and product metrics. Project metrics are then consolidated to create process metrics that are public to the software organization as a whole. But how does an organization combine metrics that come from different individuals or projects? To illustrate, consider a simple example. Individuals on two different project teams record and categorize all errors that they find during the software process. Individual measures are then combined to develop team measures. Team A found 342 errors during the software process prior to release. Team B found 184 errors. All other things being equal, which team is more effective in uncovering errors throughout the process? Because you do not know the size or complexity of the projects, you cannot answer this question. However, if the measures are normalized, it is possible to create software metrics that enable comparison to broader organizational averages.

Size-Oriented Metrics

Size-oriented software metrics are derived by normalizing quality and/or productivity measures by considering the size of the software that has been produced. If a software organization maintains simple records, a table of size-oriented measures can be created.

Function-Oriented Metrics

Function-oriented software metrics use a measure of the functionality delivered by the application as a normalization value. The most widely used function-oriented metric is the function point (FP). Computation of the function point is based on characteristics of the software's information domain and complexity.

The function point, like the LOC measure, is controversial. Proponents claim that FP is programming language independent, making it ideal for applications using conventional

and nonprocedural languages, and that it is based on data that are more likely to be known early in the evolution of a project, making FP more attractive as an estimation approach. Opponents claim that the method requires some "sleight of hand" in that computation is based on subjective rather than objective data, that counts of the information domain (and other dimensions) can be difficult to collect after the fact, and that FP has no direct physical meaning—it's just a number.

Object-Oriented Metrics

Conventional software project metrics (LOC or FP) can be used to estimate object orientedsoftware projects. However, these metrics do not provide enough granularity for the schedule and effort adjustments that are required as you iterate through an evolutionary or incremental process.

Lorenz and Kidd suggest the followingset of metrics for Object Oriented projects:

Number of scenario scripts:

A scenario script is a detailed sequence of steps that describe theinteraction between the user and the application. Each script is organized into triplets of the form

{initiator, action, participant}

where initiator is the object that requests some service (that initiates a message), action is the result of the request, and participant is the server object that satisfies the request. The number of scenario scripts is directly correlated to the size of the application and to the number of test cases that must be developed to exercise the system once it is constructed.

Number of key classes: Key classes are the "highly independent components" that are defined early in object-oriented analysis . Because key classes are central to the problem domain, the number of such classes is an

indication of the amount of effort required to develop the software and also an indicationof the potential amount of reuse to be applied during system development.

Number of support classes: Support classes are required to implement the system but are not immediately related to the problem domain. Examples might be user interface (GUI) classes, database access and manipulation classes, and computation classes. In addition, support classes can be developed for each of the key classes. Support classes are defined iteratively throughout an evolutionary process. The number of support classes is an indication of the amount of effort required to develop the software and also an indication of the potential amount of reuse to be applied during system development.

Average number of support classes per key class: In general, key classes areknown early in the project. Support classes are defined throughout. If the average number of support classes per key class were known for a given problem domain, estimating (based on total number of classes) would be greatly simplified. Lorenz and Kidd suggest that applications with a GUI have between two and three times the number of support classes as key classes. Non-GUI applications have between one and two times the number of support classes as key classes.

Number of subsystems: A subsystem is an aggregation of classes that support a function that is visible to the end user of a system. Once subsystems are identified, it is easier to lay out a reasonable schedule in which work on subsystems is partitioned among project staff.

Use-Case–Oriented Metrics: Use cases are used widely as a method for describing customer-level or business domain requirements that imply software features and functions. It would seem reasonable to use the use case as a normalization measure similar to LOC or FP.

SOFTWARE PROJECT ESTIMATION

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

1. Delay estimation until late in the project (obviously, we can achieve 100 percentaccurate estimates after the project is complete!).
2. Base estimates on similar projects that have already been completed.
3. Use relatively simple decomposition techniques to generate project cost andeffort estimates.
4. Use one or more empirical models for software cost and effort estimation.

Unfortunately, the first option, however attractive, is not practical. Cost estimatesmust be provided up-front. However, you should recognize that the longer you wait,the more you know, and the more you know, the less likely you are to make seriouserrors in your estimates.The second option can work reasonably well, if the current

project is quite similarto past efforts and other project influences (e.g., the customer, business conditions,the software engineering environment, deadlines) are roughly equivalent. Unfortunately,past experience has not always been a good indicator of future results.The remaining options are viable approaches to software project estimation.Ideally, the techniques noted for each option should be applied in tandem; each usedas a cross-check for the other.

Decomposition techniques

Software project estimation is a form of problem solving, and in most cases, the problem to be solved (i.e., developing a cost and effort estimate for a software project) is too complex to be considered in one piece. For this reason, you should decompose the problem, recharacterizing it as a set of smaller (and hopefully, more manageable) problems.But before an estimate can be made, you must understand the scope of the software to be built and generate an estimate of its "size."

Software Sizing

The accuracy of a software project estimate is predicated on a number of things:

(1) the degree to which you have properly estimated the size of the product to be built;

(2) the ability to translate the size estimate into human effort, calendar time, and dollars

(a function of the availability of reliable software metrics from past projects);

(3) the degree to which the project plan reflects the abilities of the software team; and

(4) the stability of product requirements and the environment that supports the software

engineering effort.

Putnam and Myers suggest four different approaches to the sizing problem:

• "Fuzzy logic" sizing. This approach uses the approximate reasoning techniques that are the cornerstone of fuzzy logic. To apply this approach, the planner must identify the type of application, establish its magnitude on a qualitative scale, and then refine the magnitude within the original range.

Function point sizing. The planner develops estimates of the information domain characteristics

Standard component sizing. Software is composed of a number of different "standard components" that are generic to a particular application area. For example, the standard components for an information system are subsystems, modules, screens, reports, interactive programs, batch programs, files, LOC, and object-level instructions. The project planner estimates the number of occurrences of each standard component and then uses historical project data to estimate the delivered size per standard component.

Change sizing. This approach is used when a project encompasses the use of existing software that must be modified in some way as part of a project. The planner estimates the number and type (e.g., reuse, adding code, changing code, deleting code) of modifications that must be accomplished.

Problem-Based Estimation

LOC and FP data are used in two ways during software project estimation:

(1) as estimation variables to "size" each element of the software and

(2) as baseline metrics collected from past projects and used in conjunction with estimation variables to develop cost and effort projections.

LOC and FP estimation are distinct estimation techniques. Yet both have a number of characteristics in common. You begin with a bounded statement of software scope and from this statement attempt to decompose the statement of scope into problem functions that can each be estimated individually. LOC or FP (the estimation variable) is then estimated for each function. Alternatively, you may choose another component for sizing, such as classes or objects, changes, or business processes affected.

The LOC and FP estimation techniques differ in the level of detail required for decomposition

and the target of the partitioning. When LOC is used as the estimation variable, decomposition is absolutely essential and is often taken to considerable levels of detail. The greater the degree of partitioning, the more likely reasonably accurate estimates of LOC can be developed. For FP estimates, decomposition works differently. Rather than focusing on function, each of the information domain characteristics—inputs, outputs, data files, inquiries, and external interfaces are estimated. The resultant estimates can then be used to derive an FP value that can be tied to past data and used to generate an estimate.

Regardless of the estimation variable that is used, you should begin by estimating a range of values for each function or information domain value. Using historical data or (when all else fails) intuition, estimate an optimistic, most likely, and pessimistic size value for each function or count for each information domain value. An implicit indication of the degree of uncertainty is provided when a range of values is specified. A three-point or expected value can then be computed. The expected value for the estimation variable (size) S can be computed as a weighted average of the optimistic (s_{opt}), most likely (s_m), and pessimistic (s_{pess}) estimates.

For example,

$$S = \frac{s_{opt} + 4s_m + s_{pess}}{6}$$

COCOMO - COnstructiveCOstMOdel. The original COCOMO model became one of the most widely used and discussed software cost estimation models in the industry. It has evolved intoa more comprehensive estimation model, called COCOMOII . Like its predecessor, COCOMO II is actually a hierarchy of estimation models that address the following areas:

• Application composition model. Used during the early stages of software engineering,

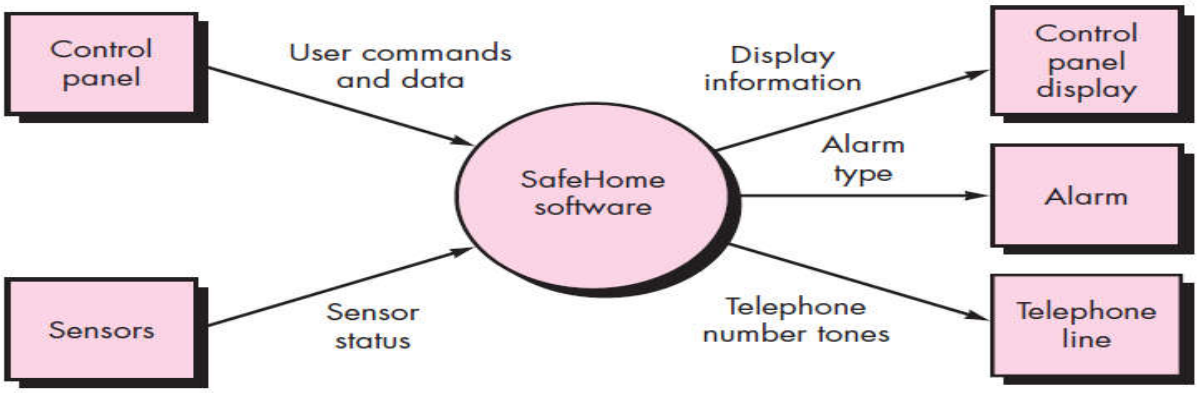
when prototyping of user interfaces, consideration of software and system interaction, assessment of performance, and evaluation of technology maturity are paramount.

• Early design stage model. Used once requirements have been stabilized and basic software architecture has been established.

• Post-architecture-stage model. Used during the construction of the software. Like all estimation models for software, the COCOMO II models require sizing information. Three different sizing options are available as part of the model hierarchy: object points, function points, and lines of source code.

Context model

The fundamental system modelor context diagram depicts the security function as a single transformation, representingthe external producers and consumers of data that flow into and out of thefunction. Figure below depicts a level 0 context model



Context-level DFD for the SafeHome security function

The behavioral model

The behavioral model indicates how software will respond to external events or stimuli. To create the model, you should perform the following steps:

1. Evaluate all use cases to fully understand the sequence of interaction within
the system.
2. Identify events that drive the interaction sequence and understand how theseevents relate to specific objects.
3. Create a sequence for each use case.
4. Build a state diagram for the system.

5. Review the behavioral model to verify accuracy and consistency.

Identifying Events with the Use Case

In general, an event occurs whenever the system and an actor exchange information.

State Representations

In the context of behavioral modeling, two different characterizations of states must be considered: (1) the state of each class as the system performs its function and

(2) the state of the system as observed from the outside as the system performs its function.

The state of a class takes on both passive and active characteristics.

A passive state is simply the current status of all of an object's attributes.

State diagrams for analysis classes.

One component of a behavioral model is a UML state diagram that represents active states for each class and the events (triggers) that cause changes between these active states. Figure below illustrates a state diagram for the Control Panel object in the SafeHome security function.

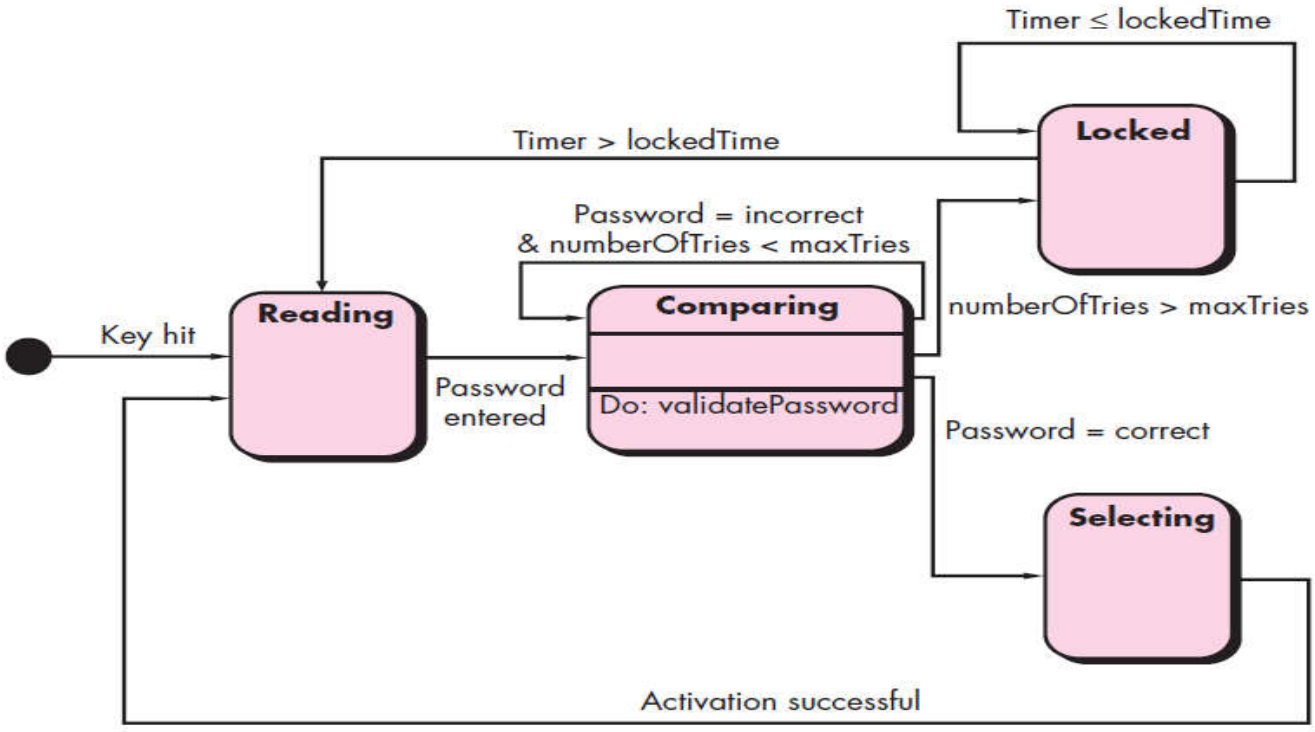


Figure : State diagram for the ControlPanel class

Sequence diagrams. The second type of behavioral representation, called a sequence diagram in UML, indicates how events cause transitions from object to object. Once events have been identified by examining a use case, the modeler creates a sequence diagram—a representation of how events cause flow from one object to another as a function of time. In essence, the sequence diagram is a shorthand version of the use case. It represents key classes and the events that cause

behavior to flow from class to class.

DATA MODEL

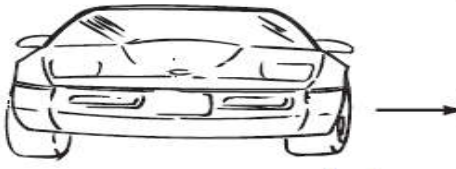
If software requirements include the need to create, extend, or interface with a database or if complex data structures must be constructed and manipulated, the software team may choose to create a data model as part of overall requirements modeling. A software engineer or analyst defines all data objects that are processed within the system, the relationships between the data objects, and other information that is pertinent to the relationships. The entity-relationship diagram (ERD) addresses these issues and represents all data objects that are entered, stored, transformed, and produced within an application.

Data Objects

A data object is a representation of composite information that must be understood by software. By composite information, I mean something that has a number of different properties or attributes. Therefore, width (a single value) would not be a valid data object, but dimensions (incorporating height, width, and depth) could be defined as an object.

Data Attributes

Data attributes define the properties of a data object and take on one of three different characteristics. They can be used to (1) name an instance of the data object, (2) describe the instance, or (3) make reference to another instance in another table. In addition, one or more of the attributes must be defined as an identifier—that is, the identifier attribute becomes a "key" when we want to find an instance of the data object. In some cases, values for the identifier(s) are unique, although this is not a requirement. Referring to the data object car, a reasonable identifier might be the ID number.



Naming attributes

Ties one data object to another, in this case, owner

Identifier

Descriptive attributes

Referential attributes

Make	Model	ID#	Body type	Color	Owner
Lexus	LS400	AB123. . .	Sedan	White	RSP
Chevy	Corvette	X456. . .	Sports	Red	CCD
BMW	750iL	XZ765. . .	Coupe	White	LJL
Ford	Taurus	Q12A45. . .	Sedan	Blue	BLF

Instance

Relationships

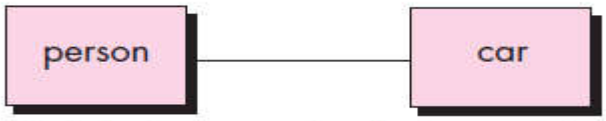
Data objects are connected to one another in different ways. Consider the two data objects, person and car. These objects can be represented using the simple notation illustrated in Figure below. A connection is established between person and car because the two objects are related. But what are the relationships? To determine the

answer, you should understand the role of people (owners, in this case) and cars within the context of the software to be built. You can establish a set of object/relationship pairs that define the relevant relationships.

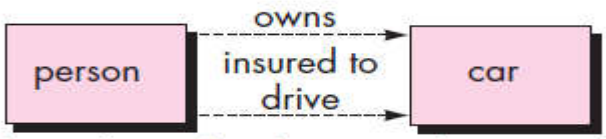
For example,

• A person owns a car.

• A person is insured to drive a car.



(a) A basic connection between data objects



(b) Relationships between data objects

Design Process

Software design is an iterative process through which requirements are translatedinto a blueprint" for constructing the software. Initially, the blueprint depicts a holisticview of software. That is, the design is represented at a high level of abstraction—a level that can be directly traced to the specific system objective and more detaileddata, functional, and behavioral requirements. As design iterations occur, subsequentrefinement leads to design representations at much lower levels of abstraction.These can still be traced to requirements, but the connection is more subtle.

Design Quality

Throughout the design process, the quality of the evolving design is assessed with a series of technical reviews. In order to evaluate the quality of a design representation, you and other members of the software team must establish technical criteria for good design.

Quality Guidelines

1. A design should exhibit an architecture that has been created using recognizable architectural styles or patterns is composed of componentsthat exhibit good design characteristics and can be implemented in an evolutionary fashion,2 thereby facilitating implementation and testing.
2. A design should be modular; that is, the software should be logically partitionedinto elements or subsystems.
3. A design should contain distinct representations of data, architecture, interfaces, and components.

4. A design should lead to data structures that are appropriate for the classes tobe implemented and are drawn from recognizable data patterns.
5. A design should lead to components that exhibit independent functionalcharacteristics.
6. A design should lead to interfaces that reduce the complexity of connectionsbetween components and with the external environment.
7. A design should be derived using a repeatable method that is driven by informationobtained during software requirements analysis.
8. A design should be represented using a notation that effectively communicatesits meaning.

Cohesion and coupling

Cohesionis an indication of the relative functional strength of a module. *Coupling* is an indication of the relative interdependence among modules. Cohesion is a natural extension of the information-hiding concept. A cohesive module performs a single task, requiring little interaction with other components in other parts of a program. Stated simply, a cohesive module should (ideally) do just one thing. Although you should always strive for high cohesion (i.e., single-mindedness), it is often necessary and advisable to have a software component perform multiple functions. However, "schizophrenic" components (modules that perform many unrelated functions) are to be avoided if a good design is to be achieved.

Coupling is an indication of interconnection among modules in a software structure.Coupling depends on the interface complexity between modules, the point atwhich entry or reference is made to a module, and what data pass across the interface.In software design, you should strive for the lowest possible coupling. Simpleconnectivity among modules results in software that is easier to understand and lessprone to a "ripple effect", caused when errors occur at one location and propagatethroughout a system.

Object Oriented Concepts

Requirements modeling (also called analysis modeling) focuses primarily on classes that are extracted directly from the statement of the problem. These entity classes typically represent things that are to be stored in a database and persist throughout the duration of the application (unless they are specifically deleted). Design refines and extends the set of entity classes. Boundary and controller classes are developed and/or refined during design. Boundary classes create the interface (e.g., interactive screen and printed reports) that the user sees and interactswith, as the software is used. Boundary classes are designed with the responsibility of managing the way entity objects are represented to users.

Controller classes are designed to manage:

(1) the creation or update of entityobjects

(2) the instantiation of boundary objects as they obtain information from entity objects,

(3) complex communication between sets of objects, and

(4) validation of data communicated between objects or between the user and the application.

The concepts discussed in the paragraphs that follow can be useful in analysis and design work.

Inheritance.

Inheritance is one of the key differentiators between conventional and object-oriented systems. A subclass Y inherits all of the attributes and operations associated with its superclass X. This means that all data structures and algorithms originally designed and implemented for X are immediately available for Y—no further work need be done. Reuse has been accomplished directly. Any change to the attributes or operations contained within a superclass is immediately inherited by all subclasses. Therefore, the class hierarchy becomes a mechanism through which changes (at high levels) can be immediately propagated through a system. It is important to note that at each level of the class hierarchy new attributes and operations may be added to those that have been inherited from higher levels in the hierarchy. In fact, whenever a new class is to be created, you have a number of options:

• The class can be designed and built from scratch. That is, inheritance is not used.

• The class hierarchy can be searched to determine if a class higher in the hierarchy contains most of the required attributes and operations. The new class inherits from the higher class and additions may then be added, as required.

• The class hierarchy can be restructured so that the required attributes and operations can be inherited by the new class.

• Characteristics of an existing class can be overridden, and different versions of attributes or operations are implemented for the new class.

Like all fundamental design concepts, inheritance can provide significant benefit for the design, but if it is used inappropriately, it can complicate a design unnecessarily and lead to error-prone software that is difficult to maintain.

Messages.

Classes must interact with one another to achieve design goals. A message stimulates some behavior to occur in the receiving object. The behavior is accomplished when an operation is executed.

Polymorphism. *Polymorphism* is a characteristic that greatly reduces the effort required to extend the design of an existing object-oriented system. To understand polymorphism, consider a conventional application that must draw four different types of graphs: line graphs, pie charts, histograms, and Kiviat diagrams. Ideally, once data are collected for a particular type of graph, the graph should draw itself.To accomplish this in a conventional application (and maintain module cohesion), it would be necessary to develop drawing modules for each type of graph.

UML

The Unified Modeling Language (UML) is “a standard language for writingsoftware blueprints. UML may be used to visualize, specify, construct, anddocument the artifacts of a software-intensive system”. In otherwords, just as building architects create blueprints to be used by a constructioncompany, software architects create UML diagrams to help software developersbuild the software.

To model classes, including their attributes, operations, and their relationships and associations with other classes, UML provides a class diagram. A class diagram provides a static or structural view of a system. It does not show the dynamic nature of the communications between the objects of the classes in the diagram. The main elements of a class diagram are boxes, which are the icons used to represent classes and interfaces. Each box is divided into horizontal parts. The top part contains the name of the class. The middle section lists the attributes of the class. An attribute refers to something that an object of that class knows or can provide all the time. Attributes are usually implemented as fields of the class, but they need not be. They could be values that the class can compute from its instance variables or values that the class can get from other objects of which it is composed.

For example, an object might always know the current time and be able to return it to you whenever you ask. Therefore, it would be appropriate to list the current time as an attribute of that class of objects. However, the object would most likely not have that time stored in one of its instance variables, because it would need to continually update that field. Instead, the object would likely compute the current time (e.g., through consultation with objects of other classes) at the moment when the time is requested. The third section of the class diagram contains the operations or behaviors of the class. An operation refers to what objects of the class can do. It is usually implemented as a method of the class.

Figure A1 presents a simple example of a Thoroughbred class that models thoroughbred horses. It has three attributes displayed—mother, father, and birthyear. The diagram also shows three operations: getCurrentAge(), getFather(), and getMother(). There may be other suppressed attributes and operations not shown in the diagram.

Figure A1 : A class diagram for a Thoroughbred class

Each attribute can have a name, a type, and a level of visibility. The type and visibility are optional. The type follows the name and is separated from the name by a colon. The visibility is indicated by a preceding –, #, ~, or +, indicating, respectively, private, protected, package, or public visibility. In Figure A1, all attributes have private visibility, as indicated by the leading minus sign (–). You can also specify that an attribute is a static or class

attribute by underlining it. Each operation can also be displayed with a level of visibility, parameters with names and types, and a return type. An abstract class or abstract method is indicated by the use of italics for the name in the class diagram. See the Horse class in Figure A2 for an example. An interface is indicated by adding the phrase "«interface»" (called a stereotype) above the name. See the OwnedObject interface in Figure A2. An interface can also be represented graphically by a hollow circle. It is worth mentioning that the icon representing a class can have other optional parts. For example, a fourth section at the bottom of the class box can be used to list the responsibilities of the class. This fourth section is not shown in any of the figures in this appendix. Class diagrams can also show relationships between classes. A class that is a subclass of another class is connected to it by an arrow with a solid line for its shaft and with a triangular hollow arrowhead. The arrow points from the subclass to the superclass. In UML, such a relationship is called a generalization. For example, in Figure A2, the Thoroughbred and QuarterHorse classes are shown to be subclasses of the Horse abstract class. An arrow with a dashed line for the arrow shaft indicates implementation of an interface. In UML, such a relationship is called a realization. For example, in Figure A2, the Horse class implements or realizes the OwnedObject interface.

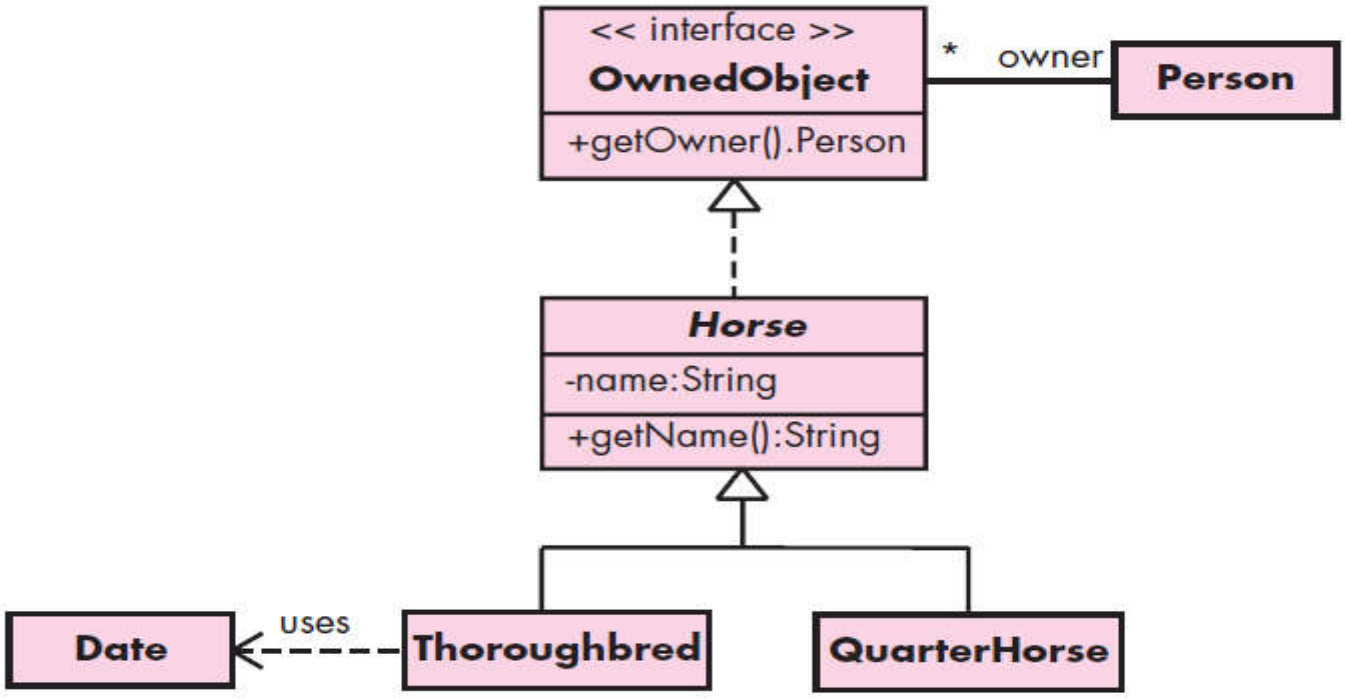


Figure A2: A class diagram regarding horses

An ***association*** between two classes means that there is a structural relationship between them. Associations are represented by solid lines. An association has many optional parts. It can be labeled, as can each of its ends, to indicate the role of each class in the association. For example, in Figure A2, there is an association between **OwnedObject**and **Person** in which the **Person** plays the role of owner. Arrows on either or both ends of an

association line indicate navigability. Also, each end of the association line can have a multiplicity value displayed. Navigability and multiplicity are explained in more detail later in this section. An association might also connect a class with itself, using a loop. Such an association indicates the connection of an object of the class with other objects of the same class. An association with an arrow at one end indicates one-way navigability. The arrow means that from one class you can easily access the second associated class to which the association points, but from the second class, you cannot necessarily easily access the first class. Another way to think about this is that the firs class is aware of the second class, but the second class object is not necessarily directly aware of the first class. An association with no arrows usually indicates a two-way association, which is what was intended in Figure A2, but it could also just mean that the navigability is not important and so was left off.

It should be noted that an attribute of a class is very much the same thing as an association of the class with the class type of the attribute. That is, to indicate that a class has a property called "name" of type String, one could display that property as an attribute, as in the **Horse** class in Figure A2. Alternatively, one could create a one-way association from the **Horse** class to the **String** class with the role of the **String** class being "name." The attribute approach is better for primitive data types, whereas the association approach is often better if the property's class plays a major role in the design, in which case it is valuable to have a class box for that type. A *dependency* relationship represents another connection between classes and is indicated by a dashed line (with optional arrows at the ends and with optional labels). One class depends on another if changes to the second class might require changes to the first class. An association from one class to another automatically indicates a dependency. No dashed line is needed between classes if there is already an association between them.

However, for a transient relationship (i.e., a class that does not maintain any long-term connection to another class but does use that class occasionally) we should draw a dashed line from the first class to the second. For example, in Figure A2, the **Thoroughbred** class uses the **Date** class whenever its *getCurrentAge()* method is invoked, and so the dependency is labeled "uses." The *multiplicity* of one end of an association means the number of objects of that class associated with the other class. A multiplicity is specified by a nonnegative integer or by a range of integers. A multiplicity specified by "0..1" means that there are 0 or 1 objects on that end of the association. For example, each person in the world has either a Social Security number or no such number (especially if they are not U.S. citizens), and so a multiplicity of 0..1 could be used in an association between a**Person** class and a **SocialSecurityNumber**class in a class diagram. A multiplicityspecified by "1..*" means one or more, and a multiplicity specified by "0..*" or just "*"means zero or more. An * was used as the multiplicity on the **OwnedObject**end ofthe association with class **Person** in Figure A2 because a **Person** can own zero ormore objects..

If one end of an association has multiplicity greater than 1, then the objects of the class referred to at that end of the association are probably stored in a collection, such as a set or ordered list. One could also include that

collection class itself in the UML diagram, but such a class is usually left out and is implicitly assumed to be there due to the multiplicity of the association.

An***aggregation***is a special kind of association indicated by a hollow diamond on one end of the icon. It indicates a "whole/part" relationship, in that the class to which the arrow points is considered a "part" of the class at the diamond end of the association.

A ***composition***is an aggregation indicating strong ownership of the parts. In a composition, the parts live and die with the owner because they have no role in the software system independent of the owner.

UML *use-case diagram* help you determine the functionality and features of the software from the user's perspective. To give you a feeling for how use cases and use-case diagrams work, we will create some for a software application for managing digital music files, similar to Apple's iTunes software.

Some of the things the software might do include:

• Download an MP3 music file and store it in the application's library.

• Capture streaming music and store it in the application's library.

• Manage the application's library (e.g., delete songs or organize them in playlists).

• Burn a list of the songs in the library onto a CD.

• Load a list of the songs in the library onto an iPod or MP3 player.

• Convert a song from MP3 format to AAC format and vice versa.

A use-case diagram for the digital music application is shown in Figure A3.

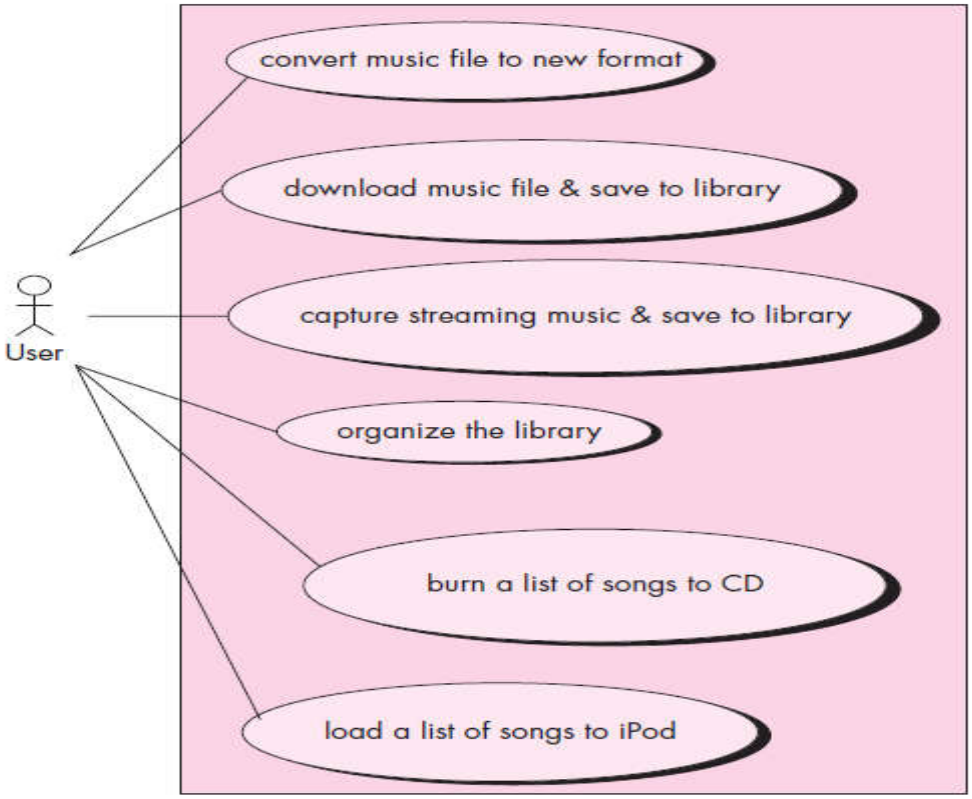


Figure A3: Use case diagram for music system

RISK ANALYSIS

Software development is activity that uses a variety of technological advancements and requires high levels of knowledge. Because of these and other factors, every software development project contains elements of uncertainty. This is known as project risk. The success of a software development project depends quite heavily on the amount of risk that corresponds to each project activity. As a project manager, it's not enough to merely be aware of the risks. To achieve a successful outcome, project leadership must identify, assess, prioritize, and manage all of the major risks. Risk is the possibility of suffering loss, and total risk exposure to a specific project will account for both the *probability* and the *size* of the potential loss.

Risk management

Risk management means risk containment and mitigation. First, you've got to identify and plan. Then be ready to act when a risk arises, drawing upon the experience and knowledge of the entire team to minimize the impact to the project.

Risk management includes the following tasks:

- ***Identify*** risks and their triggers
- ***Classify*** and prioritize all risks
- Craft a ***plan*** that links each risk to a mitigation
- ***Monitor*** for risk triggers during the project
- Implement the ***mitigating action*** if any risk materializes
- ***Communicate*** risk status throughout project

Identify and Classify Risks

Most software engineering projects are inherently risky because of the variety potential problems that might arise. Experience from other software engineering projects can help managers classify risk. The importance here is not the elegance or range of classification, but rather to precisely identify and describe all of the real threats to project success. A simple but effective classification scheme is to arrange risks according to the areas of impact.

Five Types of Risk In Software Project Management

For most software development projects, we can define five main risk impact areas:

- New, unproven technologies
- User and functional requirements
- Application and system architecture
- Performance
- Organizational

New, unproven technologies. The majority of software projects entail the use of new technologies. Ever-changing tools, techniques, protocols, standards, and development systems increase the probability that technology risks will arise in virtually any substantial software engineering effort. Training and knowledge are of critical importance, and the improper use of new technology most often leads directly to project failure.

User and functional requirements. Software requirements capture all user needs with respect to the software system features, functions, and quality of service. Too often, the process of requirements definition is lengthy, tedious, and complex. Moreover, requirements usually change with discovery, prototyping, and integration activities. Change in elemental requirements will likely propagate throughout the entire project, and modifications to user requirements might not translate to functional requirements. These disruptions often lead to one or more critical failures of a poorly-planned software development project.

Application and system architecture. Taking the wrong direction with a platform, component, or architecture can have disastrous consequences. As with the technological risks, it is vital that the team includes experts who understand the architecture and have the capability to make sound design choices.

Performance. It's important to ensure that any risk management plan encompasses user and partner expectations on performance. Consideration must be given to benchmarks and threshold testing throughout the project to ensure that the work products are moving in the right direction.

Organizational. Organizational problems may have adverse effects on project outcomes. Project management must plan for efficient execution of the project, and find a balance between the needs of the development team and the expectations of the customers. Of course, adequate staffing includes choosing team members with skill sets that are a good match with the project.

Risk Management Plan

After cataloging all of the risks according to type, the software development project manager should craft a risk management plan. As part of a larger, comprehensive project plan, the risk management plan outlines the response that will be taken for each risk—if it materializes.

Monitor and Mitigate

To be effective, software risk monitoring has to be integral with most project activities. Essentially, this means frequent checking during project meetings and critical events.

Monitoring includes:

- Publish project status reports and include risk management issues
- Revise risk plans according to any major changes in project schedule
- Review and reprioritize risks, eliminating those with lowest probability
- Brainstorm on potentially new risks after changes to project schedule or scope

When a risk occurs, the corresponding mitigation response should be taken from the risk management plan.

Mitigating options include:

- **Accept:** Acknowledge that a risk is impacting the project. Make an explicit decision to accept the risk without any changes to the project. Project management approval is mandatory here.
- **Avoid:** Adjust project scope, schedule, or constraints to minimize the effects of the risk.
- **Control:** Take action to minimize the impact or reduce the intensification of the risk.
- **Transfer:** Implement an organizational shift in accountability, responsibility, or authority to other stakeholders that will accept the risk.
- **Continue Monitoring:** Often suitable for low-impact risks, monitor the project environment for potentially increasing impact of the risk.

Communicate

Throughout the project, it's vital to ensure effective communication among all stakeholders, managers, developers, QA—especially marketing and customer representatives. Sharing information and getting feedback about risks will greatly increase the probability of project success.

UNIT-III

SYLLABUS

KARPAGAM ACADEMY OF HIGHER EDUCATION

CLASS:IIM.Com(CA)

COURSE NAME: SOFTWARE MODELS AND ENGINEERING

COURSECODE:18CCP301

BATCH-2018-2020

S/W Requirements, S/W Metrices& Testing Strategies: S/W Requirements : Functional and non-functional requirements, User requirements, System requirements.SRA& SRS. S/W Metrices: Process Metrices, Project Metrices& Product Metrices. Testing Strategies : A strategic approach to software testing, Testing fundamentals, Test Case Design. Types Of Testing: Black-Box Testing, White-Box Testing, Validation testing, System testing, the art of Debugging. Code walkthrough and reviews. Software Quality, Metrics for Analysis Model, Metrics for Design Model, Metrics for source code, Metrics for testing, Metrics for maintenance.

Requirements engineering (RE) is the process of establishing the services that the customer requires from a system and the constraints under which it operates and is developed. The **requirements** themselves are the descriptions of the system services and constraints that are generated during the requirements engineering process. Requirements may range from a high-level abstract statement of a service or of a system constraint to a detailed mathematical functional specification. As much as possible, requirements should describe **what** the system should do, but **not how** it should do it.

A functional requirement describes *what* **a software system should do, while non-functional requirements place constraints on** *how* **the system will do so.**

In software engineering, a functional requirement defines a system or its component. It describes the functions a software must perform. A function is nothing but inputs, its behavior, and outputs. It can be a calculation, data manipulation, business process, user interaction, or any other specific functionality which defines what function a system is likely to perform.

Functional software requirements help you to capture the intended behavior of the system. This behavior may be expressed as functions, services or tasks or which system is required to perform.

Functional requirements

The functional requirements for a system describe what the system should do. These requirements depend on the type of software being developed, the expected users of the software, and the general approach taken by the organization when writing requirements. When expressed as user requirements, functional requirements are usually described in an abstract way that can be understood by system users. However, more specific functional system requirements describe the system functions, its inputs and outputs, exceptions, etc., in detail. Functional system requirements vary from general requirements covering what the system should do to very specific requirements reflecting local ways of working or an organization's existing systems.

For example, here are examples of functional requirements for the MHC-PMS system, used to maintain information about patients receiving treatment for mental health problems:

1. A user shall be able to search the appointments lists for all clinics.

2. The system shall generate each day, for each clinic, a list of patients who are expected to attend appointments that day.

3. Each staff member using the system shall be uniquely identified by his or her eight-digit employee number.

These functional user requirements define specific facilities to be provided by the system. These have been taken from the user requirements document and they show that functional requirements may be written at different levels of detail (contrast requirements 1 and 3). Imprecision in the requirements specification is the cause of many software engineering problems. It is natural for a system developer to interpret an ambiguous requirement in a way that simplifies its implementation. Often, however, this is not what the customer wants. New requirements have to be established and changes made to the system. Of course, this delays system delivery and increases costs. For example, the first example requirement for the MHC-PMS states that a user shall be able to search the appointments lists for all clinics. The rationale for this requirement is that patients with mental health problems are sometimes confused. They may have an appointment at one clinic but actually go to a different clinic. If they have an appointment, they will be recorded as having attended, irrespective of the clinic. The medical staff member specifying this may expect 'search' to mean that, given a patient name, the system looks for that name in all appointments at all clinics. However, this is not explicit in the requirement. System developers may interpret the requirement in a different way and may implement a search so that the user has to choose a clinic then carry out the search. This obviously will involve more user input and so take longer. In principle, the functional requirements specification of a system should be both complete and consistent. Completeness means that all services required by the user should be defined. Consistency means that requirements should not have contradictory definitions. In practice, for large, complex systems, it is practically impossible to achieve

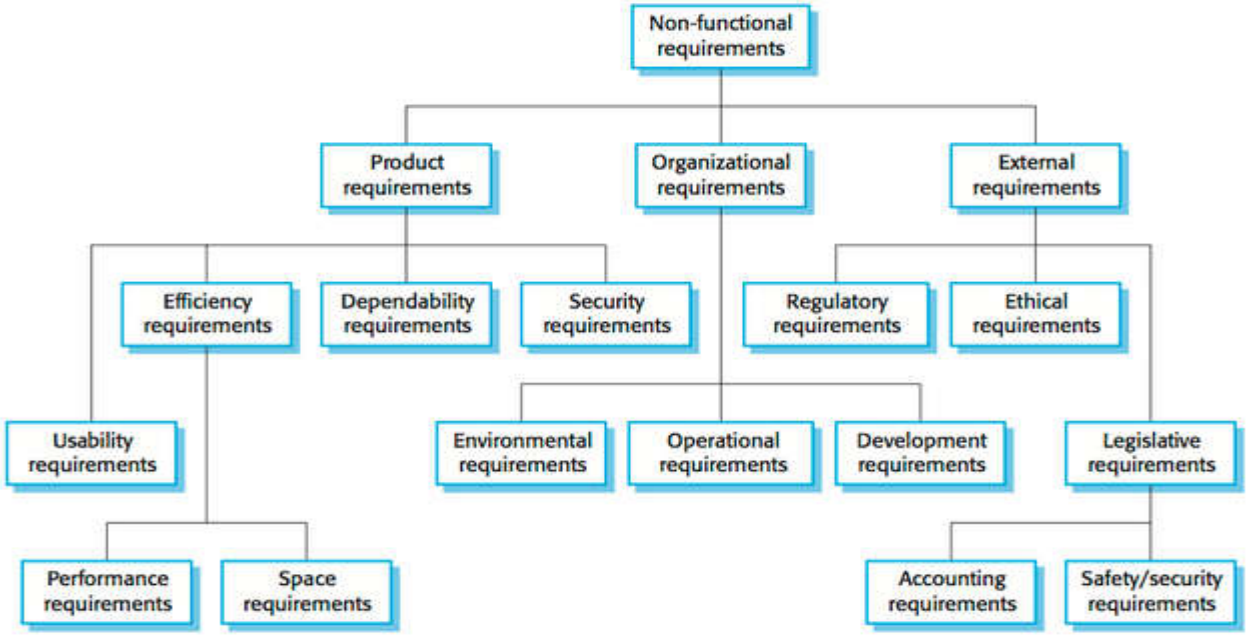
requirements consistency and completeness. One reason for this is that it is easy to make mistakes and omissions when writing specifications for complex systems. Another reason is that there are many stakeholders in a large system. A stakeholder is a person or role that is affected by the system in some way. Stakeholders have different—and often inconsistent—needs. These inconsistencies may not be obvious when the requirements are first specified, so inconsistent requirements are included in the specification. The problems may only emerge after deeper analysis or after the system has been delivered to the customer.

Non Functional Requirements

Non-functional requirements, as the name suggests, are requirements that are not directly concerned with the specific services delivered by the system to its users. They may relate to emergent system properties such as reliability, response time, and store occupancy. Alternatively, they may define constraints on the system implementation such as the capabilities of I/O devices or the data representations used in interfaces with other systems. Non-functional requirements, such as performance, security, or availability, usually specify or constrain characteristics of the system as a whole. Non-functional requirements are often more critical than individual functional requirements. System users can usually find ways to work around a system function that doesn't really meet their needs. However, failing to meet a non-functional requirement can mean that the whole system is unusable. For example, if an aircraft system does not meet its reliability requirements, it will not be certified as safe for operation; if an embedded control system fails to meet its performance requirements, the control functions will not operate correctly. Although it is often possible to identify which system components implement specific functional requirements (e.g., there may be formatting components that implement reporting requirements), it is often more difficult to relate components to non-functional requirements. The implementation of these requirements may be diffused throughout the system. There are two reasons for this:

1. Non-functional requirements may affect the overall architecture of a system rather than the individual components. For example, to ensure that performance requirements are met, you may have to organize the system to minimize communications between components.

2. A single non-functional requirement, such as a security requirement, may generate a number of related functional requirements that define new system services that are required. In addition, it may also generate requirements that restrict existing requirements.



Types of non-functional requirement

Three classes of non-functional requirements:

1. Product requirements

Requirements which specify that the delivered product must behave in a particular way e.g. execution speed, reliability, etc.

2. Organizational requirements

Requirements which are a consequence of organizational policies and procedures e.g. process standards used, implementation requirements, etc.

3. External requirements

Requirements which arise from factors which are external to the system and its development process e.g. interoperability requirements, legislative requirements, etc.

Non-functional requirements may be very difficult to state precisely and imprecise requirements may be difficult to verify. If they are stated as a **goal** (a general intention of the user such as ease of use), they should be rewritten as a **verifiable** non-functional requirement (a statement using some quantifiable metric that can be objectively tested). Goals are helpful to developers as they convey the intentions of the system users.

User requirements

High-level abstract requirements written as statements, in a natural language plus diagrams, of what services the system is expected to provide to system users and the constraints under which it must operate. The **user**

requirement(s) document (URD) or user requirement(s) specification (URS) is a document usually used in software engineering that specifies what the user expects the software to be able to do. Once the required information is completely gathered it is documented in a URD, which is meant to spell out exactly what the software must do and becomes part of the contractual agreement. A customer cannot demand features not in the URD, whilst the developer cannot claim the product is ready if it does not meet an item of the URD. The URD can be used as a guide to planning cost, timetables, milestones, testing, etc. The explicit nature of the URD allows customers to show it to various stakeholders to make sure all necessary features are described. Formulating a URD requires negotiation to determine what is technically and economically feasible. Preparing a URD is one of those skills that lies between a science and an art, requiring both software technical skills and interpersonal skills.

The user requirements for a system should describe the functional and nonfunctional requirements so that they are understandable by system users who don't have detailed technical knowledge. Ideally, they should specify only the external behavior of the system. The requirements document should not include details of the system architecture or design. Consequently, if you are writing user requirements, you should not use software jargon, structured notations, or formal notations. You should write user requirements in natural language, with simple tables, forms, and intuitive diagrams.

User requirements are almost always written in natural language supplemented by appropriate diagrams and tables in the requirements document. System requirements may also be written in natural language but other notations based on forms, graphical system models, or mathematical system models can also be used. Graphical models are most useful when you need to show how a state changes or when you need to describe a sequence of actions. UML sequence charts and state charts show the sequence of actions that occur in response to a certain message or event. Formal mathematical specifications are sometimes used to describe the requirements for safety- or security-critical systems, but are rarely used in other circumstances.

System requirements

System requirements are expanded versions of the user requirements that are used by software engineers as the starting point for the system design. They add detail and explain how the user requirements should be provided by the system. They may be used as part of the contract for the implementation of the system and should therefore be a complete and detailed specification of the whole system. Ideally, the system requirements should simply describe the external behavior of the system and its operational constraints. They should not be concerned with how the system should be designed or implemented. However, at the level of detail required to completely specify a complex software system, it is practically impossible to exclude all design information. There are several reasons for this:

1. You may have to design an initial architecture of the system to help structure the requirements specification. The system requirements are organized according to the different sub-systems that make up the system.
2. In most cases, systems must interoperate with existing systems, which constrain the design and impose requirements on the new system.
3. The use of a specific architecture to satisfy non-functional requirements may be necessary. An external regulator who needs to certify that the system is safe may specify that an already certified architectural design be used.

Software requirements specification (SRS)

It is a document that describes what the software will do and how it will be expected to perform. An SRS describes the functionality the product needs to fulfill all stakeholders (business, users) needs. A software requirements specification (SRS) is a document that captures complete description about how the system is expected to perform. It is usually signed off at the end of requirements engineering phase.

A software requirements specification (SRS) is a document that is created when a detailed description of all aspects of the software to be built must be specified before the project is to commence. It is important to note that a formal SRS is not always written. In fact, there are many instances in which effort expended on an SRS might be better spent in other software engineering activities. However, when software is to be developed by a third party, when a lack of specification would create severe business issues, or when a system is extremely complex or business critical, an SRS may be justified.

Quality Characteristics of a good SRS

Correctness:

User review is used to ensure the correctness of requirements stated in the SRS. SRS is said to be correct if it covers all the requirements that are actually expected from the system.

Completeness:

Completeness of SRS indicates every sense of completion including the numbering of all the pages, resolving the to be determined parts to as much extent as possible as well as covering all the functional and non-functional requirements properly.

Consistency:

Requirements in SRS are said to be consistent if there are no conflicts between any set of requirements. Examples

of conflict include differences in terminologies used at separate places, logical conflicts like time period of report generation, etc.

 Unambiguousness: An SRS is said to be unambiguous if all the requirements stated have only 1 interpretation. Some of the ways to prevent unambiguousness include the use of modelling techniques like ER diagrams, proper reviews and buddy checks, etc.

 Ranking for importance and stability: There should a criterion to classify the requirements as less or more important or more specifically as desirable or essential. An identifier mark can be used with every requirement to indicate its rank or stability.

 Modifiability: SRS should be made as modifiable as possible and should be capable of easily accepting changes to the system to some extent. Modifications should be properly indexed and cross-referenced.

 Verifiability: An SRS is verifiable if there exists a specific technique to quantifiably measure the extent to which every requirement is met by the system. For example, a requirement stating that the system must be user-friendly is not verifiable and listing such requirements should be avoided.

 Traceability:
One should be able to trace a requirement to a design component and then to a code segment in the program. Similarly, one should be able to trace a requirement to the corresponding test cases.

 Design Independence:
There should be an option to choose from multiple design alternatives for the final system. More specifically, the SRS should not include any implementation details.

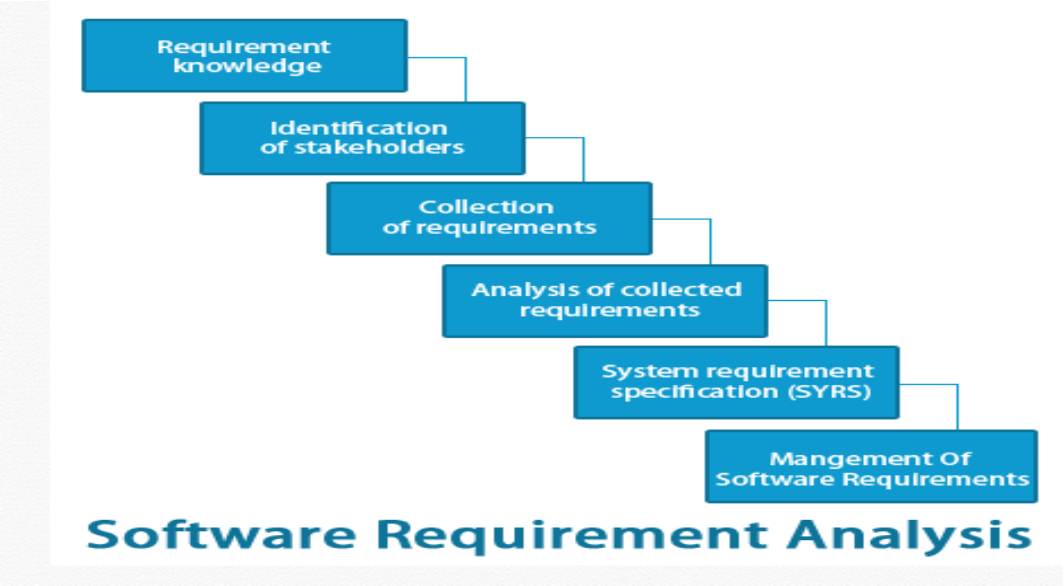
 Testability:
An SRS should be written in such a way that it is easy to generate test cases and test plans from the document.

 Understandable by the customer:
An end user maybe an expert in his/her specific domain but might not be an expert in computer science. Hence, the use of formal notations and symbols should be avoided to as much extent as possible. The language should be kept easy and clear.

 Right level of abstraction:
If the SRS is written for the requirements phase, the details should be explained explicitly. Whereas, for a feasibility study, fewer details can be used. Hence, the level of abstraction varies according to the purpose of the

Software Requirement Analysis(SRA)

Software requirement is a functional or non-functional need to be implemented in the system. Functional means providing particular service to the user. For example, in context to banking application the functional requirement will be when customer selects "View Balance" they must be able to look at their latest account balance. Software requirement can also be a non-functional, it can be a performance requirement. For example, a non-functional requirement is where every page of the system should be visible to the users within 5 seconds.



Necessity Of Requirement Analysis

According to statistics major reason of failure of software is that it does not meet with the requirement of the user. Requirement analysis involves the task that determines the needs of the software, which mainly includes complaints and needs of various clients/stakeholders.

Software Requirement Analysis Process

The steps for effective capturing on present requirements of users are:

- Requirement Knowledge:

It is very necessary to know about the requirements of the users before starting any project. Working on the present requirements of the users will be helpful in gaining popularity of your project.

- Identification of Stakeholders:

Stakeholders includes customers, end-users, system administrators etc. identifying the correct stakeholder is second step and is one of the most important step in all. Identifying the correct stakeholders help to properly analyze and create a road map for gathering requirements.

- Collection of Requirements:

After identifying stakeholders one has to collect requirements for them. Based on the nature and aim of the project there can be many kinds of stakeholders. Interacting with stakeholder groups can be in person interviews, focus groups, market study, surveys and secondary research.

- Analysis of Collected Requirements:

Once the data is gathered structured analysis must be done of the data to make models. Data are analysed on the basis of various parameters depending on the goals of the software. These include animation, automated reasoning, knowledge based critiquing, consistency checking, analogical and case based reasoning.

- System requirement Specification (SYRS):

Once the data is analyzed they are put together in the form of system requirement specification document (SYRS) or system requirement specification (SRS). It acts as a blueprint for the designing team to make the project. It serves as a technical collection of all the requirements of stake holders which includes user requirements, system requirements, user interface and operational requirements.

- Management Of Software Requirements:

The last step of this analysis process is correcting and validating all elements of requirement specifications document. Errors can be corrected at this stage. Minor changes can also be done according to the requirement of the software user.

Code Walkthrough is a form of peer review in which a programmer leads the review process and the other team members ask questions and spot possible errors against development standards and other issues.

- The meeting is usually led by the author of the document under review and attended by other members of the team.
- Review sessions may be formal or informal.
- Before the walkthrough meeting, the preparation by reviewers and then a review report with a list of findings.
- The scribe, who is not the author, marks the minutes of meeting and note down all the defects/issues so that it can be tracked to closure.
- The main purpose of walkthrough is to enable learning about the content of the document under review to help team members gain an understanding of the content of the document and also to find defects.

UNIT IV

SYLLABUS

KARPAGAM ACADEMY OF HIGHER EDUCATION

CLASS:IIM.Com(CA)

COURSE NAME: SOFTWARE MODELS AND ENGINEERING

COURSECODE:18CCP301

BATCH-2018-2020

Testing Plan and Maintenance: Snooping for information, Coping with complexity through teaming, Testing plan focus areas, Testing for recoverability, Planning for troubles, Preparing for the tests: Software Reuse, Developing good test programs , Data corruption, Tools, Test Execution ,Testing with a virtual computer, Simulation and Prototypes, Managing the Test, Customer's role in testing, Software maintenance issues and techniques. Software reuse. Client-Server software development.

Snooping for information

Snooping, in a security context, is unauthorized access to another person's or company's data. The practice is similar to eavesdropping but is not necessarily limited to gaining access to data during its transmission. Snooping can include casual observance of an e-mail that appears on another's computer screen or watching what someone else is typing. More sophisticated snooping uses software programs to remotely monitor activity on a computer or network device.

Malicious hackerkeyloggers to monitor keystrokes, capture passwords and login information, and to intercept e-mail and other private communications and data transmissions. Corporations sometimes snoop on employees legitimately to monitor their use of business computers and track Internet usage; governments may snoop on individuals to collect information and avert crime and terrorism.

Test plan focus area

A **TEST PLAN** is a document describing software testing scope and activities. It is the basis for formally testing any software/product in a project.

- **test plan**: A document describing the scope, approach, resources and schedule of intended test activities. It identifies amongst others test items, the features to be tested, the testing tasks, who will do each task, degree of tester independence, the test environment, the test design techniques and entry and exit criteria to be used, and the rationale for their choice,and any risks requiring contingency planning. It is a record of the test planning process.
- **master test plan**: A test plan that typically addresses multiple test levels.
- **phase test plan**: A test plan that typically addresses one test phase.

Test Plan Types

One can have the following types of test plans:

- **Master Test Plan:** A single high-level test plan for a project/product that unifies all other test plans.
- **Testing Level Specific Test Plans:**Plans for each level of testing.
    - Unit Test Plan
    - Integration Test Plan
    - System Test Plan
    - Acceptance Test Plan
- **Testing Type Specific Test Plans:** Plans for major types of testing like Performance Test Plan and Security Test Plan.

Recovery testing in software testing

Recovery testing is a type of non-functional **testing** technique performed in order to determine how quickly the system can **recover** after it has gone through system crash or hardware failure. **Recovery testing** is the forced failure of the **software** to verify if the **recovery** is successful.It involves reverting to a point where the integrity of the system was known and then reprocessing transactions up to the point of failure.

The purpose of recovery testing is to verify the system's ability to recover from varying points of failure.

The time taken to recover depends upon:

- The number of restart points
- A volume of the applications
- Training and skills of people conducting recovery activities and tools available for recovery.

When there are a number of failures then instead of taking care of all failures, the recovery testing should be done in a structured fashion which means recovery testing should be carried out for one segment and then another.

It is done by professional testers. Before recovery testing, adequate backup data is kept in secure locations. This is done to ensure that the operation can be continued even after a disaster.

Life Cycle of Recovery Process

The life cycle of the recovery process can be classified into the following five steps:

1. Normal operation
2. Disaster occurrence
3. Disruption and failure of the operation
4. Disaster clearance through the recovery process
5. Reconstruction of all processes and information to bring the whole system to move to normal operation

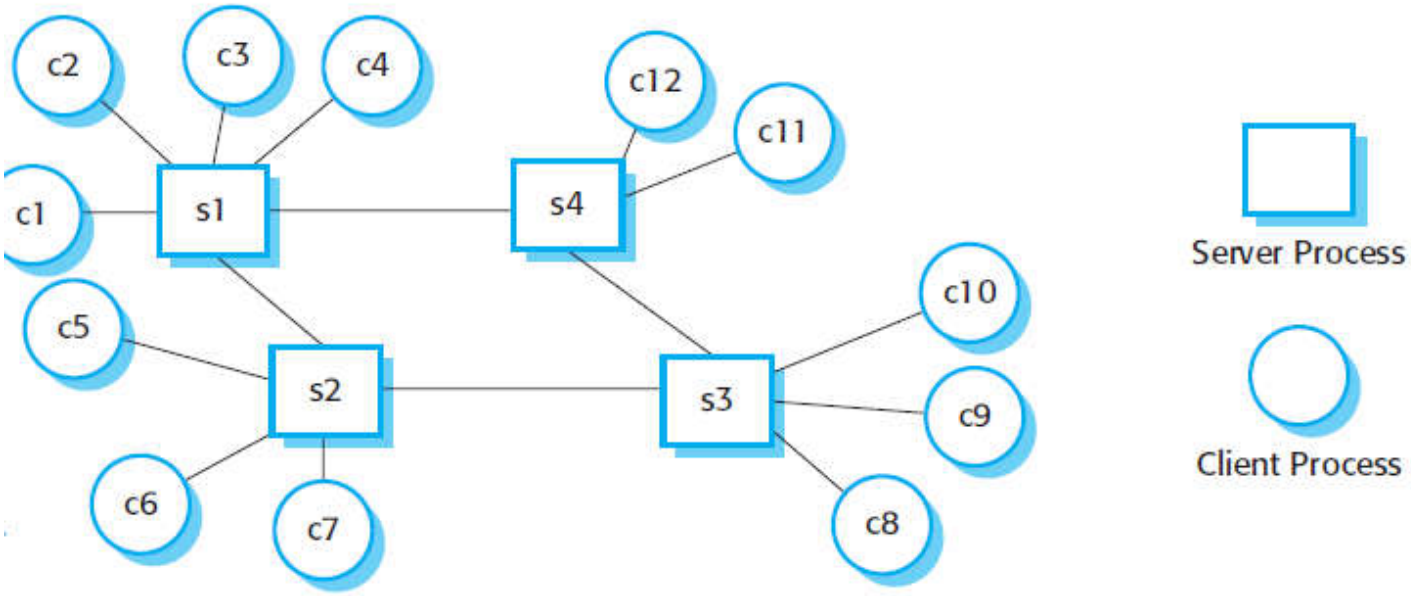
1. A system consisting of hardware, software, and firmware integrated to achieve a common goal is made operational for carrying out a well-defined and stated goal. The system is called to perform the normal operation to carry out the designed job without any disruption within a stipulated period of time.
2. A disruption may occur due to malfunction of the software, due to various reasons like input initiated malfunction, software crashing due to hardware failure, damaged due to fire, theft, and strike.
3. Disruption phase is a most painful phase which leads to business losses, relation break, opportunity losses, man-hour losses and invariably financial and goodwill losses. Every sensible agency should have a plan for disaster recovery to enable the disruption phase to be minimal.
4. If a backup plan and risk mitigation processes are at the right place before encountering disaster and disruption, then recovery can be done without much loss of time, effort and energy. A designated individual, along with his team with the assigned role of each of these persons should be defined to fix the responsibility and help the organization to save from long disruption period.
5. Reconstruction may involve multiple sessions of operation to rebuild all folders along with configuration files. There should be proper documentation and process of reconstruction for correct recovery.

While performing recovery testing following things should be considered.

- We must create a test bed as close to actual conditions of deployment as possible. Changes in interfacing, protocol, firmware, hardware, and software should be as close to the actual condition as possible if not the same condition.
- Through exhaustive testing may be time-consuming and a costly affair, identical configuration, and complete check should be performed.
- If possible, testing should be performed on the hardware we are finally going to restore. This is especially true if we are restoring to a different machine than the one that created the backup.
- Some backup systems expect the hard drive to be exactly the same size as the one the backup was taken from.
- Obsolescence should be managed as drive technology is advancing at a fast pace, and old drive may not be compatible with the new one. One way to handle the problem is to restore to a virtual machine. Virtualization software vendors like VMware Inc. can configure virtual machines to mimic existing hardware, including disk sizes and other configurations.
- Online backup systems are not an exception for testing. Most online backup service providers protect us from being directly exposed to media problems by the way they use fault-tolerant storage systems.
- While online backup systems are extremely reliable, we must test the restore side of the system to make sure there are no problems with the retrieval functionality, security or encryption.

Client server softwaredevelopment

If protection of data is a critical requirement, then a client–server architecture should be used, with the protection mechanisms built into the server. However, if the protection is compromised, then the losses associated with an attack are likely to be high, as are the costs of recovery (e.g., all user credentials may have to be reissued). The system is vulnerable to denial of service attacks, which overload the server and make it impossible for anyone to access the system database.



Client server interaction

client–server model

This is a multi-user, web-based system for providing a film and photograph library. In this system, several servers manage and display the different types of media. Video frames need to be transmitted quickly and in synchrony but at relatively low resolution. They may be compressed in a store, so the video server can handle video compression and decompression in different formats. Still pictures, however, must be maintained at a high resolution, so it is appropriate to maintain them on a separate server. The catalog must be able to deal with a variety of queries and provide links into the web information system that includes data about the film and video clips, and an e-commerce system that supports the sale of photographs, film, and video clips.

UNIT-V

SYLLABUS

KARPAGAM ACADEMY OF HIGHER EDUCATION

CLASS:IIM.Com(CA)

COURSE NAME: SOFTWARE MODELS AND ENGINEERING

COURSECODE:18CCP301

BATCH-2018-2020

Software Reengineering and Project Management: Software Reengineering, Reverse Engineering & Forward Engineering, Life Cycle Phases and Process artifacts, Restructuring. Model based software architectures, Software process and Iteration workflows, Major and Minor milestones, Periodic status assessments, Process Planning, Project Control and process instrumentation: Seven core metrics, management indicators, quality indicators, life-cycle expectations, CCPDS-R Case Study and Future Software Project Management Practices.

Techniques for Maintenance

To perform software maintenance effectively, various techniques are used. These include software configuration management, impact analysis, and software rejuvenation, all of which help in maintaining a system and thus, improve the quality of the existing system.

Software Configuration Management

Software configuration management can be used effectively while maintaining a system as it keeps track of changes and their effects on the system components. Many changes occur when the software is delivered to the users such as failure or users' request for enhancement in the software. For this, configuration control board (CCB) oversees the entire change process. Note that the representatives of CCB along with the users and developers manage changes collectively. These changes are managed in the following steps.

1. When the user encounters a problem such as failure report, he requests for change on a formal change request form. The problem can also be an enhancement to a function, variation in the older function, or deleting an existing function. The procedure for request of change remains the same. The change reqest form should include information about how the system works, nature of the problem, and how the new (expected) system should work.
2. The request for change is reported to CCB.
3. The representative of CCB meets the user to discuss the problem (That is, to determine that the request is for failure report or for enhancement).

4. If the user requests for a reported failure, the CCB discusses the source of the problem. If the requested change is an enhancement, the CCB discusses the parts or the components that will be affected by the change. In both the cases, developers describe the scope of changes and the expected time to implement them.
5. The developers determine the source of the problem or the components which will be affected when the changes will be implemented. For this, they use a test copy instead of the operational system and implement the requested changes to see whether it (test copy) performs according to the requested changes.
6. Finally, after the changes have been made, all the relevant documentation is updated according to the requested change.
7. The developers then record all the changes made to the operational system in a change report to keep track of the next release or version of the software system.

Version control implies the process by which the contents of the software, hardware, or documentation are revised. It tracks and manages the progress of files and directories within a project. This process is required when one or more components of a software system are changed (for example, Microsoft has introduced MSN Messenger 7.0, which is an upgraded version of MSN Messenger 6.2). Software maintenance manages the versions, that is, the older version (present software) and the new version (when the software is modified). Note that the software configuration management manages how the versions differ, who made the changes, and why they were made.

The component (existing version) is assigned an identification number. When the version (current) is revised, a revision number is allotted to each resulting changed component. The records such as name of the component, date and time, version status, and account of all changes are managed. This helps the software configuration management to identify the current version and the revised number of the operational system.

Impact Analysis

Impact analysis is used to evaluate the overall effect of the requested change. This includes identifying the components that will be affected with the change, the extent to which each of the components will be affected, and the consequences of change on the estimated effort and schedule. There are various advantages of performing impact analysis, which are listed below.

1. It is used to understand the situations when the modifications required in the software system affect large segments of software code or several components of the software.
2. It helps identifying the relationship among the components that are affected with the change and thereby helping to understand the overall software structure.
3. It is used to record the history of modification, which helps in maintaining quality in the software system.

Software Rejuvenation

Sometimes, organizations have to take difficult decisions about how to make their systems more maintainable. The choices may include enhancing or completely replacing a software system. Note that each choice has the same objective, that is, to preserve or increase the software quality while keeping the costs low. Software rejuvenation is a maintenance technique which helps in taking appropriate decisions.

Software rejuvenation checks the system's work products in order to extract additional information or to reformat them in order to make these work products more understandable. Generally, four types of software rejuvenation exist, namely, re-documentation, restructuring, reverse engineering, and reengineering. Re-documentation uses static analysis of the source code to produce additional information, which helps the software maintenance team to understand and refer to the code. In source code, component size, component calls, calling parameters, and control

paths are examined to understand what and how code does it. The output of static code analysis is either graphical or textual, which can be used to assess whether the re-documentation is required.

Restructuring

Restructuring involves the transformation of unstructured code into structured code thereby making it easier to understand and change. Restructuring involves the following steps.

1. Static analysis is performed, which provides information that is used to represent code as a directed graph or associative (semantic) network. The representation mayor may not be in a human readable form; thus, an automated tool is used.
2. Transformational techniques are used to refine (simplify) the representation.
3. Refined representation is interpreted and used to generate the structured code.

Reverse Engineering

Reverse engineering like re-documentation, focuses on providing information about the specification and design information using the software code. The information extracted from specification and design is stored in a format that can be easily modified. Reverse engineering is a useful technique when the software maintenance team is unable to understand the processes involved in the software system. Reverse engineering involves the following steps.

1. Source code is collected with the help of an automated tool used for reverse engineering. This tool is used to represent the structure and the naming information of variables, functions and other components in the software code.
2. Static analysis is performed.
3. Some methods such as standards structured analysis and design methods are used. These methods are used to extract information such as data dictionaries, data-flow, control flow, and entity relationship (ER) diagrams for the reverse engineering technique.

The advantages associated with reverse engineering are listed below.

1. It focuses on recovering the lost information from the programs.
2. It provides the abstract information from the detailed source code implementation.
3. It improves system documentation 'that is either incomplete or out of date.
4. It manages the complexity that is present in the software programs.
5. It detects the adverse effects of modification in the software system.

Re-engineering

Re-engineering is an extension of reverse engineering. This technique refers to the systematic transformation of the present software system into a new form to make quality improvements in operation, system capability, functionality, and achieving high performance at low costs.

Re-engineering involves the following steps.

1. The system is reverse engineered and represented internally for human and computer modifications.
2. The software system is corrected and completed. This includes updating internal specification and design.
3. Using new specification and design, a new system is generated.

Advantages

Reduced cost: Generally, it is observed that the software systems that are maintained using re-engineering incur less cost as compared to developing the software system all over again.

Reduced risk: The incremental nature of re-engineering means that the existing staff skills evolve as the software system evolves. Due to this fact, the risks associated with the modifications in the software system are reduced.

Better use of existing staff: The individuals who worked on software maintenance can be retained while the re-engineering technique is being used. In addition, the staff can be extended to accommodate new skills during reengineering. Due to this fact, the re-engineering technique has less number of risks and incurs less expenditure while hiring the new staff.

Incremental development: Re-engineering techniques can be carried out in stages according to the availability of budget and resources. This technique is useful in operational organizations with working software systems. In such organizations, the staff can easily adapt to the re-engineered software system.

Data corruption

Data corruption refers to errors in computerdata that occur during writing, reading, storage, transmission, or processing, which introduce unintended changes to the original data. Computer, transmission, and storage systems use a number of measures to provide end-to-end data integrity, or lack of errors.

In general, when data corruption occurs a file containing that data will produce unexpected results when accessed by the system or the related application. Results could range from a minor loss of data to a system crash. For example, if a document file is corrupted, when a person tries to open that file with a document editor they may get an error message, thus the file might not be opened or might open with some of the data corrupted (or in some cases, completely corrupted, leaving the document unintelligible). The adjacent image is a corrupted image file in which most of the information has been lost.

Some types of malware may intentionally corrupt files as part of their payloads, usually by overwriting them with inoperative or garbage code, while a non-malicious virus may also unintentionally corrupt files when it accesses them. If a virus or trojan with this payload method manages to alter files critical to the running of the computer's operating system software or physical hardware, the entire system may be rendered unusable.

Some programs can give a suggestion to repair the file automatically (after the error), and some programs cannot repair it. It depends on the level of corruption, and the built-in functionality of the application to handle the error.

Causes of data corruption and loss

Common causes of data corruption and loss include:

- Power outages or other power-related problems.
- Improper shutdowns, such as caused by power outages or performing a hard restart: pressing and holding the power button or, on Macs so equipped, the restart button.
- Hardware problems or failures, including hard drive failures, bad sectors, bad RAM, and the like.
- Failure to eject external hard drives and related storage devices before disconnecting them or powering them off.
- Bad programming, particularly if it results in either hard restarts or data that is saved incorrectly.

Any of these causes can result in a corrupted hard drive directory. A corrupted hard drive directory can cause files to apparently "go missing" and lead to further data loss or corruption, such files being overwritten with new data as a corrupted directory may no longer accurately reflect what disk space is free or available vs. the disk space that contains data. The term *data* is used here to mean both files you have created as well as application and operating system code. Technologies such as File System Journaling have helped to reduce the potential for directory corruption due to power outages or hard restarts, but journaling is not foolproof. Likewise, while hard drives have become exceedingly reliable, they are still known to fail catastrophically with little or no warning. If operating system files become corrupted, your Mac may not start up or experience recurring kernel panics when a corrupted kernel extension is used. In addition to the causes cited above, operating system files can be corrupted by failed Software Updates. Accordingly, it is appropriate to implement certain strategies to minimize the risks of data corruption and loss.

KARPAGAM ACADEMY OF HIGHER EDUCATION

CLASS:IIM.Com(CA)

COURSE NAME: SOFTWARE MODELS AND ENGINEERING

COURSECODE:18CCP301

BATCH-2018-2020

UNIT-I

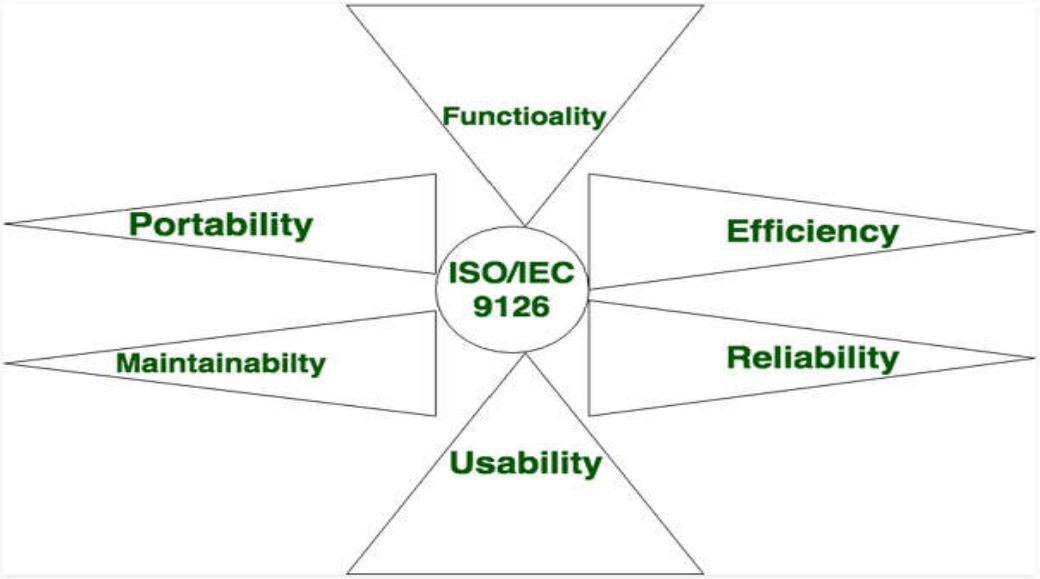
SYLLABUS

Fundamentals of Software Engineering and Process models : Definition, Software characteristics and Application. Software myths, Software engineering- A layered technology and SDLC. Software process models: Linear sequential model, prototyping model, RAD Model. Evolutionary process models: Incremental process models and Spiral model. Component based ,4GT. Maturity Models: CMM, CMMI, PCMM, PSP, TSP, Process patterns, process assessment. Unified process: SEI CMM and ISO 9001. PSP and Six Sigma. Clean room technique.

Software Engineering definition proposed by Fritz Bauer at the seminal conference on the subject still serves as a basis for discussion:

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

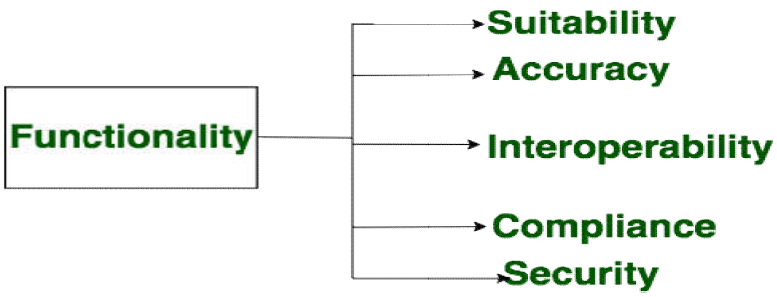
Software is defined as collection of computer programs, procedures, rules and data. Software Characteristics are classified into six major components:



These components are described below:

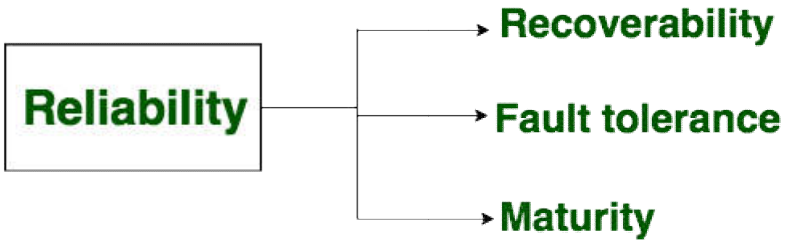
Functionality:

It refers to the degree of performance of the software against its intended purpose. Required functions are:



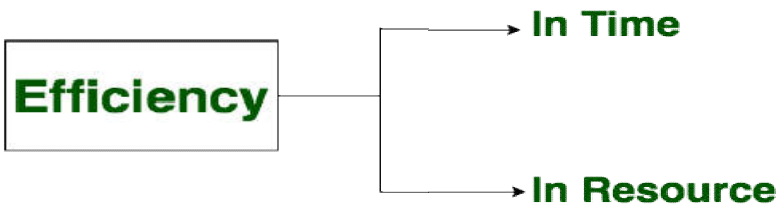
Reliability:

A set of attribute that bear on capability of software to maintain its level of performance under the given condition for a stated period of time. Required functions are:



- **Efficiency:**

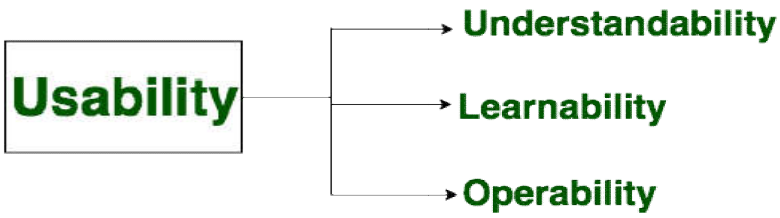
  It refers to the ability of the software to use system resources in the most effective and efficient manner. The software should make effective use of storage space and executive command as per desired timing requirement. Required functions are:



- **Usability:**

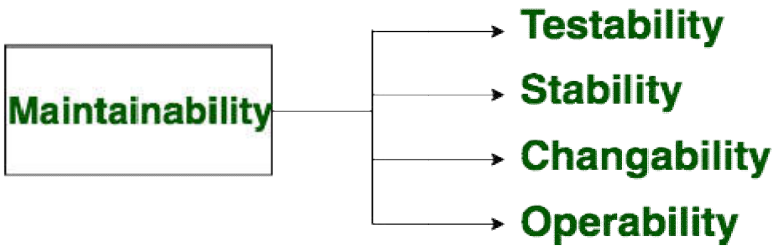
  It refers to the extent to which the software can be used with ease, the amount of effort or time required to learn how to use the software.

  Required functions are:



- **Maintainability:**

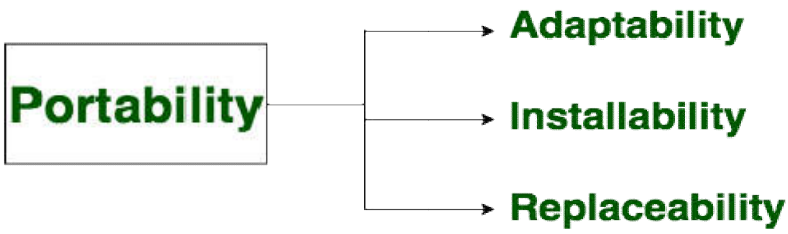
  It refers to the ease with which the modifications can be made in a software system to extend its functionality, improve its performance, or correct errors. Required functions are:



- **Portability:**

  A set of attribute that bear on the ability of software to be transferred from one environment to another, without or minimum changes.

  Required functions are:



Software myths—It is erroneous belief about software and the process that is used to build it—can be traced to the earliest days of computing. Myths have a number of attributes that make them insidious. For instance, they appear to be reasonablestatements of fact (sometimes containing elements of truth), they have an intuitive feel, and they are often promulgated by experienced practitioners who "know the score." Today, most knowledgeable software engineering professionals recognize myths for what they are—misleading attitudes that have caused serious problems for managers and practitioners alike. However, old attitudes and habits are difficult to modify, and remnants of software myths remain.

Management myths. Managers with software responsibility, like managers inmost disciplines, are often under pressure to maintain budgets, keep schedules fromslipping, and improve quality. Like a drowning person who grasps at a straw, a softwaremanager often grasps at belief in a software myth, if that belief will lessen thepressure (even temporarily).

Myth: We already have a book that's full of standards and procedures forbuilding software. Won't that provide my people with everything theyneed to know?Reality: Are softwarepractitioners aware of its existence? Does it reflect modernsoftware engineering practice? Is it complete? Is itstreamlined to improve time-to-delivery while still maintaining afocus on quality? In many cases, the answer to all of these questionsis "no."

Myth: If we get behind schedule, we can add more programmers and catch up(sometimes called the "Mongolian horde" concept).

Reality: Software development is not a mechanistic process like manufacturing.In the words of Brooks : "adding people to a late softwareproject makes it later." At first, this statement may seemcounterintuitive. However, as new people are added, people whowere working must spend time educating the newcomers, therebyreducing the amount of time spent on productive developmenteffort. People can be added but only in a planned and coordinatedmanner.

Myth: If I decide to outsource the software project to a third party, I can justrelax and let that firm build it.

Reality: If an organization does not understand how to manage and controlsoftware projects internally, it will invariably struggle when it outsourcessoftware projects.

Customer myths. A customer who requests computer software may be a personat the next desk, a technical group

down the hall, the marketing/sales department,or an outside company that has requested software under contract. In many cases,the customer believes myths about software because software managers and practitionersdo little to correct misinformation. Myths lead to false expectations (by thecustomer) and, ultimately, dissatisfaction with the developer.

Myth: A general statement of objectives is sufficient to begin writingprograms—we can fill in the details later.
Reality: Although a comprehensive and stable statement of requirements isnot always possible, an ambiguous "statement of objectives" is arecipe for disaster. Unambiguous requirements (usually derivediteratively) are developed only through effective and continuouscommunication between customer and developer.

Myth: Software requirements continually change, but change can be easilyaccommodated because software is flexible.
Reality: It is true that software requirements change, but the impact ofchange varies with the time at which it is introduced. When requirementschanges are requested early (before design or code has beenstarted), the cost impact is relatively small. However, as timepasses, the cost impact grows rapidly—resources have been committed,a design framework has been established, and change cancause upheaval that requires additional resources and major designmodification.

Practitioner's myths. Myths that are still believed by software practitioners havebeen fostered by over 50 years of programming culture. During the early days, programmingwas viewed as an art form. Old ways and attitudes die hard.

Myth: Once we write the program and get it to work, our job is done.Reality: Someone once said that "the sooner you begin 'writing code,' thelonger it'll take you to get done." Industry data indicate that between60 and 80 percent of all effort expended on software will be expendedafter it is delivered to the customer for the first time.

Myth: Until I get the program "running" I have no way of assessing its quality.
Reality: One of the most effective software quality assurance mechanismscan be applied from the inception of a project—the technical review.

Myth: The only deliverable work product for a successful project is the workingprogram.

Reality: A working program is only one part of a software configuration thatincludes many elements. A variety of work products (e.g., models, documents, plans) provide a foundation for successful engineeringand, more important, guidance for software support.

Myth: Software engineering will make us create voluminous and unnecessarydocumentation and will invariably slow us down.

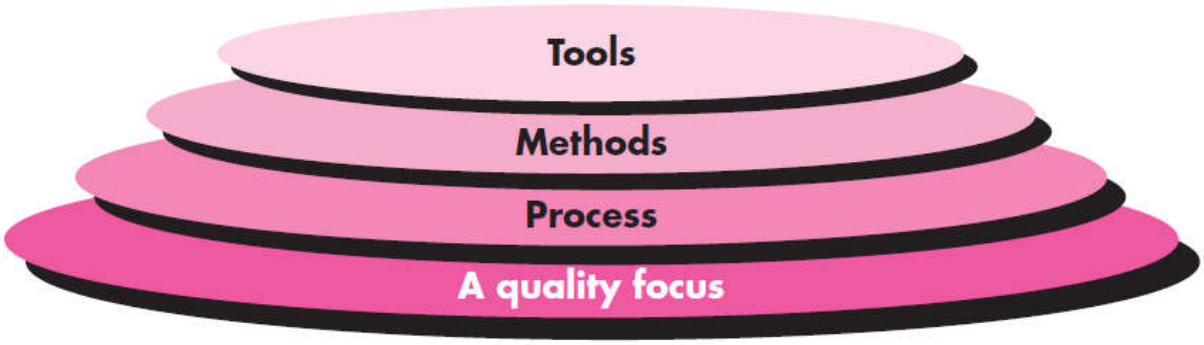
Reality: Software engineering is not about creating documents. It is aboutcreating a quality product. Better quality leads to reduced rework.And reduced rework results in faster delivery times.

Layered Technology

Software engineering is a layered technology. Referring to Figure 1.3, any engineering approach (including software engineering) must rest on an organizational commitment to quality. Total quality management, Six Sigma, and similar philosophies foster a continuous process improvement culture, and it is this culture that ultimately leads to the development of increasingly more effective approaches to software engineering. The bedrock that supports software engineering is a quality focus. The foundation for software engineering is the process layer. The software engineeringprocess is the glue that holds the technology layers together and enables rational and timely development of computer software.

FIGURE 1.3

Software engineering layers



Software engineering is Divided into **4** layers:-

1. A quality Process :-

- Any engineering approach must rest on quality.
- The "Bed Rock" that supports software Engineering is Quality Focus.

2. Process :-

- Foundation for SE is the Process Layer
- SE process is the GLUE that holds all the technology layers together and enables the timely development of computer software.

- It forms the base for management control of software project.

3. Methods :-

- SE methods provide the "Technical Questions" for building Software.
- Methods contain a broad array of tasks that include communication requirement analysis, design modeling, program construction testing and support.

4. Tools :-

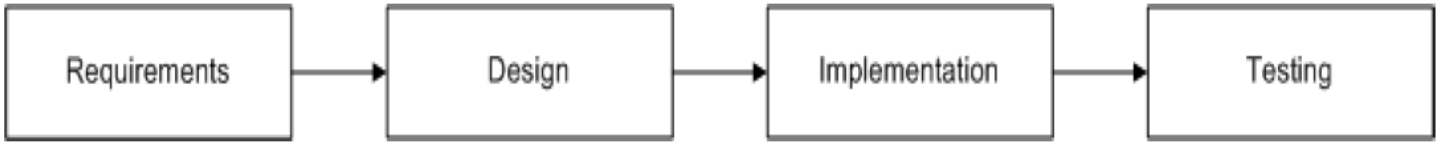
- SE tools provide automated or semi-automated support for the "Process" and the "Methods".
- Tools are integrated so that information created by one tool can be used by another.

A maturity level is a well-defined evolutionary plateau toward achieving a mature software process. Each maturity level provides a layer in the foundation for continuous process improvement.

Software Life Cycle

A Life Cycle shows how a living thing borns, grows, lives, and dies. The stages from birth to death.Software life cycle model is the stages of development that a software development goes through. The following figure shows the stages of software development.

Software life cycle models describe phases of the software cycle and the order in which those phases are executed. There are tons of models, and many companies adopt their own, but all have very similar patterns. The general, basic model is shown below:



Each phase produces deliverables required by the next phase in the life cycle. Requirements are translated into design. Code is produced during implementation that is driven by the design. Testing verifies the deliverable of the implementation phase against requirements.

A **Software Process** can be defined as set of activities, methods, practices and transformations which people employ to develop and maintain software and the associated products. The quality of a software product is essentially determined by the quality of the processes employed to develop and maintain it.

The Linear Sequential Model

This is a software process model that involves a systematic progression through **analysis**, **design**, **coding**, **testing** and **maintenance** phases. It is also referred to as the "**waterfall model**".

Also known as the classic life cycle or waterfall model, it suggests a systematic, sequential approach to software development. Problems with this approach are:

• Real projects rarely follow the sequential flow and changes can cause confusion.

• This model has difficulty accommodating requirements change

• The customer will not see a working version until the project is nearly complete

• Developers are often blocked unnecessarily, due to previous tasks not being done

The Prototyping Model

Advantages:

- Easy and quick to identify customer requirements

- Customers can validate the prototype at the earlier stage and provide their inputs and feedback

- Good to deal with the following cases:

1. Customer cannot provide the detailed requirements
2. Very complicated system-user interactions
3. Use new technologies, hardware and algorithms
4. Develop new domain application systems

Problems:

-The prototype can serve as **—the first system**.

-Developers usually attempt to develop the product based on the prototype.

-Developers often make implementation compromises in order to get a prototyping that is working quickly.

-Customers may be unaware that the prototype is not a product, which is held with.

The RAD Model

Rapid Application Development (RAD) is a linear sequential software development process model that emphasizes an extremely short development cycle.

- A —high-speed‖ adaptation of linear sequential model

- Component-based construction

- Effective when requirements are well understood and project scope is constrained.

Advantages:

- Short development time

- Cost reduction due to software reuse and component-based construction

Problems:

- For large, but scalable projects, RAD requires sufficient resources.

- RAD requires developers and customers who are committed to the schedule.
- Constructed software is project-specific, and may not be well modularized.
- Its quality depends on the quality of existing components.
- Not appropriate projects with high technical risk and new technologies.

Incremental Process Models

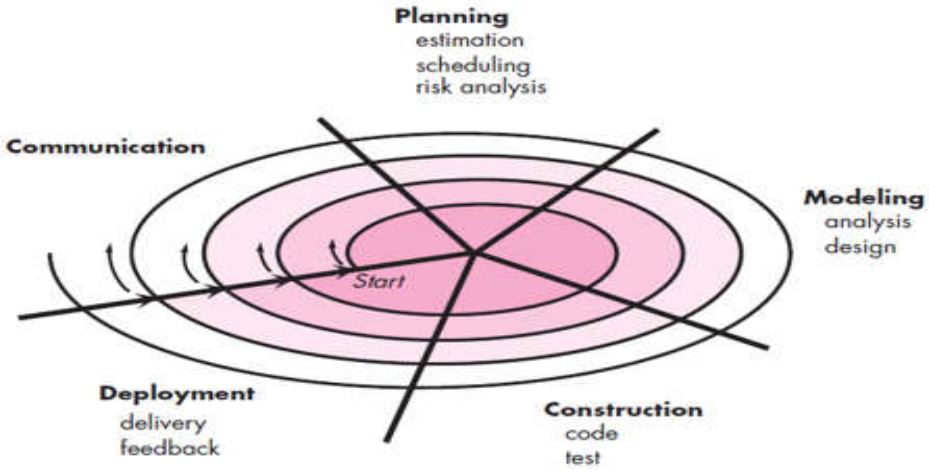
There are many situations in which initial software requirements are reasonably well defined, but the overall scope of the development effort precludes a purely linear process. In addition, there may be a compelling need to provide a limited set of software functionality to users quickly and then refine and expand on that functionalityin later software releases. In such cases, you can choose a process model that is designed to produce the software in increments. The incremental model combines elements of linear and parallel process flows

Incremental model applies linear sequences in a staggered fashion as calendar time progresses. Each linear sequence produces deliverable "increments" of the software in a manner that is similar to the increments produced by an evolutionary process flow.For example, word-processing software developed using the incremental paradigm might deliver basic file management, editing, and document production functionsin the first increment; more sophisticated editing and document production capabilities in the second increment; spelling and grammar checking in the third increment; and advanced page layout capability in the fourth increment. It should be noted that the process flow for any increment can incorporate the prototyping paradigm.

When an incremental model is used, the first increment is often a core product. That is, basic requirements are addressed but many supplementary features (some known, others unknown) remain undelivered. The core product is used by the customer (or undergoes detailed evaluation).

The Spiral Model.

Originally proposed by Barry Boehm, the spiral model is an evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the waterfall model. It providesthe potential for rapid development of increasingly more complete versions of the software. Boehm describes the model in the following manner: The spiral development model is a risk-driven process model generator that is used to guide multi-stakeholder concurrent engineering of software intensive systems. It has two main distinguishing features. One is a cyclic approach for incrementally growing a system's degree of definition and implementation while decreasing its degree of risk. The other is a set of anchor point milestones for ensuring stakeholder commitment to 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 model or prototype. During later iterations, increasingly more complete versions of the engineered system are produced.



A Typical Spiral Model

Component-based development model

Commercial off-the-shelf (COTS) software components, developed by vendors who offer them as products, provide targeted functionality with well-defined interfaces that enable the component to be integrated into the software that is to be built. The component-based development model incorporates many of the characteristics of the spiral model. It is evolutionary in nature, demanding an iterative approach to the creation of software. However, the component-based development model constructs applications from prepackaged software components. Modeling and construction activities begin with the identification of candidate components. These components can be designed as either conventional software modules or object-oriented classes or packages of classes. Regardless of the technology that is used to create the components, the component-based development model incorporates the following steps (implemented using an evolutionary approach):

1. Available component-based products are researched and evaluated for theapplication domain in question.
2. Component integration issues are considered.
3. A software architecture is designed to accommodate the components.
4. Components are integrated into the architecture.
5. Comprehensive testing is conducted to ensure proper functionality.

The component-based development model leads to software reuse, and reusability provides software engineers with a number of measurable benefits. Your software engineering team can achieve a reduction in development cycle time as well as a reduction in project cost if component reuse becomes part of your culture.

4GT Process Model

4GT begins with from “Requirement Gathering” this process go through the customer, the customer go illustrate the requirements. The customer could actually describe the requirements and these would be directly translated

into an operational prototype. If the product is a smaller product this process may be possible to move directly from requirements gathering step to implementation using a non-procedural fourth generation language (4GL), for lager products this procedure may little hard, therefor it's necessary to use the design strategy in 4GT.When it comes to large projects, the design phase it is crucial to avoid poor quality, poor maintainability. To transform a 4GL implementation into a product, the developer must conduct through testing, develop meaningful documentation, and perform all other solution integration activities. The 4GT developed software must be built in a manner that enables maintenance to be performed expeditiously. There are some merits to summarize the current features of 4GT approach. In the 4GL implementation the code can be generated based on some specification. The 4GT developed software must be built in a manner that enables maintenance to be performed expeditiously. There are some merits to summarize the current features of 4GT approach. The use of 4GT is a viable approach for many different application areas coupled with computer- aided software engineering tools and code generators, 4GT offers a credible solution to many software problem.

Benefits of the 4GT

Flexible: The Fourth Generation applications are Modifiable by Design, which means they are designed from the beginning to accommodate change. They are easily modifiable, either by you, the customer, or by your Fourth Generation Authorized reseller to your specifications.

Scalable: The Fourth Generation applications are Modifiable by Design, which means they are designed from the beginning to accommodate change.They are easily modifiable, either by you, the customer, or by your Fourth Generation Authorized reseller to your specifications.

Total Data Access: Your data represents your company's greatest single asset. The worth of that asset, however, is directly related to your ability to record it and access it.

The Fourth Generation Technique (4GT) is based on NPL that is the Non-Procedural Language techniques. Depending upon the specifications made the 4GT move towards uses various tools for the automatic generation of source codes. It is the very important tool which make use of the non-procedural language for Report generation, Database query, Manipulation of data, Interaction of screen, Definition, Generation of code, Spread Sheet capabilities, and High level graphical capacity etc. 4GT begins with a requirement-gathering stage. The customer would illustrate requirements and these would be directly converted into an unworkable operational prototype. For small applications, it may be possible to move directly from requirements gathering step to implementation using a non-procedural fourth generation language (4GL), however for large application it is necessary to develop a design strategy for the system even if a 4GL is to be used. Implementation using a 4GT enables the software developer to represent desired result in a manner that leads to automatic generation of code to create those results, obviously,

data structure with relevant information must exist and be readily accessible by the 4GL. To transform a 4GL implementation into a product, the developer must conduct through testing, develop meaningful documentation, and perform all other solution integration activities. The 4GT developed software must be built in a manner that enables maintenance to be performed expeditiously. There are some merits to summarize the current features of 4GT approach. The use of 4GT is a viable approach for many different application areas coupled with computer-aided software engineering tools and code generators, 4GT offers a credible solution to many software problem. Data collected from companies that use 4Gt indicates that the time required to produce software is greatly reduced for small and intermediate application is also reduced. However the use of 4GT for large software development efforts demands as much or more analysis design and testing to achieve substantial timesaving that result from the elimination of coding.

Compatibility Maturity Model -CMM

Maturity level 1 _Initial

organizations often produce products and services that work; however, they frequently exceed the budget and schedule of their projects.Maturity level 1 organizations are characterized by a tendency to over commit, abandon processes in the time of crisis, and not be able to repeat their past successes.

Maturity Level 2 - Managed

At maturity level 2, an organization has achieved all the specific and generic goals of the maturity level 2 process areas. In other words, the projects of the organization have ensured that requirements are managed and that processes are planned, performed, measured, and controlled.The process discipline reflected by maturity level 2 helps to ensure that existing practices are retained during times of stress. When these practices are in place, projects are performed and managed according to their documented plans.At maturity level 2, requirements, processes, work products, and services are managed. The status of the work products and the delivery of services are visible to management at defined points.Commitments are established among relevant stakeholders and are revised as needed. Work products are reviewed with stakeholders and are controlled.The work products and services satisfy their specified requirements, standards, and objectives.

Maturity Level 3 - Defined

At maturity level 3, an organization has achieved all the specific and generic goals of the process areas assigned to maturity levels 2 and 3.At maturity level 3, processes are well characterized and understood, and are described in standards, procedures, tools, and methods.A critical distinction between maturity level 2 and maturity level 3 is the scope of standards, process descriptions, and procedures. At maturity level 2, the standards, process descriptions, and procedures may be quite different in each specific instance of the process (for example, on a particular project). At maturity level 3, the standards, process descriptions, and procedures for a project are tailored from the

organization's set of standard processes to suit a particular project or organizational unit. The organization's set of standard processes includes the processes addressed at maturity level 2 and maturity level 3. As a result, the processes that are performed across the organization are consistent except for the differences allowed by the tailoring guidelines.Another critical distinction is that at maturity level 3, processes are typically described in more detail and more rigorously than at maturity level 2. At maturity level 3, processes are managed more proactively using an understanding of the interrelationships of the process activities and detailed measures of the process, its work products, and its services.

Maturity Level 4 - Quantitatively managed

At maturity level 4, an organization has achieved all the specific goals of the process areas assigned to maturity levels 2, 3, and 4 and the generic goals assigned to maturity levels 2 and 3.At maturity level 4 Sub processes are selected that significantly contribute to overall process performance. These selected sub processes are controlled using statistical and other quantitative techniques.Quantitative objectives for quality and process performance are established and used as criteria in managing processes. Quantitative objectives are based on the needs of the customer, end users, organization, and process implementers. Quality and process performance are understood in statistical terms and are managed throughout the life of the processes.

Maturity Level 5 - Optimizing

At maturity level 5, an organization has achieved all the specific goals of the process areas assigned to maturity levels 2, 3, 4, and 5 and the generic goals assigned to maturity levels 2 and 3.Processes are continually improved based on a quantitative understanding of the common causes of variation inherent in processes.Maturity level 5 focuses on continually improving process performance through both incremental and innovative technological improvements.Quantitative process-improvement objectives for the organization are established, continually revised to reflect changing business objectives, and used as criteria in managing process improvement.

Capability Maturity Model Integration - CMMI

Capability Maturity Model Integration (CMMI) is a process level improvement training and appraisal program. Administered by the CMMI Institute, a subsidiary of ISACA, it was developed at Carnegie Mellon University (CMU). It is required by many United States Department of Defense (DoD) and U.S. Government contracts, especially in software development. CMU claims CMMI can be used to guide process improvement across a project, division, or an entire organization. CMMI defines the following maturity levels for processes: Initial, Repeatable, Defined, Quantitatively Managed, and Optimizing.

CMMI Model

1) Initial: The software process is characterized as ad hoc, and occasionally even chaotic. Few processes are defined, and success depends on individual effort and heroics.

2) Repeatable: Basic project management processes are established to track cost, schedule, and functionality. The necessary process discipline is in place to repeat earlier successes on projects with similar applications.

3) Defined: The software process for both management and engineering activities is documented, standardized, and integrated into a standard software process for the organization. All projects use an approved, tailored version of the organization's standard software process for developing and maintaining software.

4) Managed: Detailed measures of the software process and product quality are collected. Both the software process and products are quantitatively understood and controlled.

5) Optimizing: Continuous process improvement is enabled by quantitative feedback from the process and from piloting innovative ideas and technologies.

People Capability Maturity Model (PCMM)

People Capability Maturity Model (PCMM) is a maturity framework that focuses on continuously improving the management and development of the human assets of a software or information systems organization. PCMM can be perceived as the application of the principles of Capability Maturity Model to human assets of a software organization. It describes an evolutionary improvement path from ad hoc, inconsistently performed practices, to a mature, disciplined, and continuously improving development of the knowledge, skills, and motivation of the workforce. Although the focus in People CMM is on software or information system organizations, the processes

and practices are applicable for any organization that aims to improve the capability of its workforce. PCMM will be guiding and effective particularly for organizations whose core processes are knowledge intensive.The primary objective of the People Capability Maturity Model is to improve the capability of the entire workforce. This can be defined as the level of knowledge, skills, and process abilities available for performing an organization's current and future business activities.

10 Principles of People Capability Maturity Model (PCMM)

The People Capability Maturity Model describes an evolutionary improvement path from ad hoc, inconsistently performed workforce practices, to a mature infrastructure of practices for continuously elevating workforce capability. The philosophy implicit the PCMM can be summarized in ten principles.In mature organizations, workforce capability is directly related to business performance.
Workforce capability is a competitive issue and a source of strategic advantage.
Workforce capability must be defined in relation to the organization's strategic business objectives.
Knowledge-intense work shifts the focus from job elements to workforce competencies.
Capability can be measured and improved at multiple levels, including individuals, workgroups, workforce competencies, and the organization.
An organization should invest in improving the capability of those workforce competencies that are critical to its core competency as a business.
Operational management is responsible for the capability of the workforce.
The improvement of workforce capability can be pursued as a process composed from proven practices and procedures.
The organization is responsible for providing improvement opportunities, while individuals are responsible for taking advantage of them.
Since technologies and organizational forms evolve rapidly, organizations must continually evolve their workforce practices and develop new workforce competencies.
The People Capability Maturity Model (People CMM) is a roadmap for implementing workforce practices that continuously improve the capability of an organization's workforce. Since an organization cannot implement all of the best workforce practices in an afternoon, the People CMM introduces them in stages. Each progressive level of the People CMM produces a unique transformation in the organization's culture by equipping it with more powerful practices for attracting, developing, organizing, motivating, and retaining its workforce. Thus, the People CMM establishes an integrated system of workforce practices that matures through increasing alignment with the organization's business objectives, performance, and changing needs.

Although the People CMM has been designed primarily for application in knowledge intense organizations, with appropriate tailoring it can be applied in almost any organizational setting. The People CMM's primary objective is to improve the capability of the workforce. Workforce capability can be defined as the level of knowledge, skills, and process abilities available for performing an organization's business activities.

Personal Software Process (PSP)

Every developer uses some process to build computer software. The process may be haphazard or ad hoc; may change on a daily basis; may not be efficient, effective, or even successful; but a "process" does exist. Watts Humphrey [Hum97] suggests that in order to change an ineffective personal process, an individual must move through four phases, each requiring training and careful instrumentation. The Personal Software Process (PSP) emphasizes personal measurement of both the work product that is produced and the resultant quality of the work product. In addition PSP makes the practitioner responsible for project planning (e.g., estimating and scheduling) and empowers the practitioner to control the quality of all software work products that are developed. The PSP model defines five framework activities:

Planning.

This activity isolates requirements and develops both size and resource estimates. In addition, a defect estimate (the number of defects projected for the work) is made. All metrics are recorded on worksheets or templates. Finally, development tasks are identified and a project schedule is created.

High-level design. External specifications for each component to be constructed are developed and a component design is created. Prototypes are built when uncertainty exists. All issues are recorded and tracked.

High-level design review. Formal verification methods are applied to uncover errors in the design. Metrics are maintained for all important tasks and work results.

Development. The component-level design is refined and reviewed. Code is generated, reviewed, compiled, and tested. Metrics are maintained for all important tasks and work results.

Postmortem. Using the measures and metrics collected (this is a substantial amount of data that should be analyzed statistically), the effectiveness of the process is determined. Measures and metrics should provide guidance formodifying the process to improve its effectiveness.

Team Software Process (TSP)

Because many industry-grade software projects are addressed by a team of practitioners, Watts Humphrey extended the lessons learned from the introduction of PSP and proposed a Team Software Process (TSP). The goal of TSP is to build a "selfdirected" project team that organizes itself to produce high-quality software.

Humphrey defines the following objectives for TSP:

• Build self-directed teams that plan and track their work, establish goals, andown their processes and plans. These

can be pure software teams or integratedproduct teams (IPTs) of 3 to about 20 engineers.

• Show managers how to coach and motivate their teams and how to helpthem sustain peak performance.

• Accelerate software process improvement by making Level 5behavior normal and expected.

• Provide improvement guidance to high-maturity organizations.

• Facilitate university teaching of industrial-grade team skills.

A self-directed team has a consistent understanding of its overall goals and objectives; defines roles and responsibilities for each team member; tracks quantitativeproject data (about productivity and quality); identifies a team process that is appropriate for the project and a strategy for implementing the process; defines local standards that are applicable to the team's software engineering work; continually assesses risk and reacts to it; and tracks, manages, and reports project status.

TSP defines the following framework activities: project launch, high-level design, implementation, integration and test, and postmortem. Like their counterparts in PSP (note that terminology is somewhat different), these activities enable the team to plan, design, and construct software in a disciplined manner while at the same time quantitatively measuring the process and the product. The postmortem sets the stage for process improvements.

PROCESS PATTERNS

Every software team encounters problems as it moves through the software process. It would be useful if proven solutions to these problems were readily available to the team so that the problems could be addressed and resolved quickly. A *process pattern* describes a process-related problem that is encountered during software engineering work, identifies the environment in which the problem has been encountered, and suggests one or more proven solutions to the problem. Stated in more general terms, a process pattern provides you with a template—a consistent method for describing problem solutions within the context of the software process. By combining patterns, a software team can solve problems and construct a process that best meets the needs of a project.

Patterns can be defined at any level of abstraction. In some cases, a pattern might be used to describe a problem (and solution) associated with a complete processmodel (e.g., prototyping). In other situations, patterns can be used to describe a problem (and solution) associated with a framework activity (e.g., **planning**) or an action within a framework activity (e.g., project estimating).

Ambler has proposed a template for describing a process pattern:

Pattern Name. The pattern is given a meaningful name describing it within the context of the software process (e.g., **TechnicalReviews**).

Forces. The environment in which the pattern is encountered and the issues that make the problem visible and may affect its solution.

Type. Ambler suggests three types of patterns:

1. Stage pattern—defines a problem associated with a framework activity for the process. Since a framework activity encompasses multiple actions andwork tasks, a stage pattern incorporates multiple task patterns (see the following) that are relevant to the stage (framework activity). An example of astage pattern might be EstablishingCommunication. This pattern would incorporate the task pattern RequirementsGathering and others.

2. Task pattern—defines a problem associated with a software engineering action or work task and relevant to successful software engineeringpractice (e.g., RequirementsGathering is a task pattern).

3. Phase pattern—define the sequence of framework activities that occurs within the process, even when the overall flow of activities is iterativein nature. An example of a phase pattern might be **SpiralModel** or **Prototyping.**

PROCESS ASSESSMENT

The existence of a software process is no guarantee that software will be delivered on time, that it will meet the customer's needs, or that it will exhibit the technical characteristics that will lead to long-term quality characteristics. Process patterns must be coupled with solid software engineering practice

(Part 2 of this book). In addition, the process itself can be assessed to ensure that it meets a set of basic process criteria that have been shown to be essential for a successful software engineering.. A number of different approaches to software process assessment and improvement have been proposed over the past few decades:

Standard CMMI Assessment Method for Process Improvement

(SCAMPI)—provides a five-step process assessment model that incorporates five phases: initiating, diagnosing, establishing, acting, and learning. The

SCAMPI method uses the SEI CMMI as the basis for assessment.

CMM-Based Appraisal for Internal Process Improvement (CBA IPI) — provides a diagnostic technique for assessing the relative maturity of asoftware organization; uses the SEI CMM as the basis for the assessment.

SPICE (ISO/IEC15504)—a standard that defines a set of requirements for software process assessment. The intent of the standard is to assist organizations

in developing an objective evaluation of the efficacy of any defined software process .

ISO 9001:2000 for Software—a generic standard that applies to any organization that wants to improve the overall quality of the products, systems,

or services that it provides. Therefore, the standard is directly applicable to software organizations and companies

Six Sigma for Software Engineering

Six Sigma is the most widely used strategy for statistical quality assurance in industry today. Originally popularized by Motorola in the 1980s, the Six Sigma strategy "is a rigorous and disciplined methodology that uses data and statistical analysis to measure and improve a company's operational performance by identifying and

eliminating defects' in manufacturing and service-related processes". The term Six Sigma is derived from six standard deviations—3.4 instances (defects) per million occurrences—implying an extremely high quality standard. The Six Sigma methodology defines three core steps:

- *Define* customer requirements and deliverables and project goals via welldefined methods of customer communication.
- *Measure* the existing process and its output to determine current quality performance (collect defect metrics).
- *Analyze* defect metrics and determine the vital few causes.

If an existing software process is in place, but improvement is required, Six Sigmasuggests two additional steps:

- *Improve* the process by eliminating the root causes of defects.
- *Control* the process to ensure that future work does not reintroduce the causes of defects.

Clean room technique (clean room design)

The clean room technique is a process in which a new product is developed by **reverse engineering** an existing product, and then the new product is designed in such a way that patent or **copyright** infringement is avoided. The clean room technique is also known as clean room design. (Sometimes the words "clean room" are merged into the single word, "cleanroom.") Sometimes this process is called the **Chinese wall** method, because the intent is to place a demonstrable intellectual barrier between the reverse engineering process and the development of the new product.

The use of the clean room technique can be compared, in some respects, with the fair use of copyrighted publications in order to compile a new document. For example, a new book about Linux can be authored on the basis of information obtained by researching existing books, articles, white papers, and Web sites. This does not necessarily constitute copyright infringement, even though other books on Linux already exist, and even if the new book contains essentially the same information as the existing publications. However, this is the case only as long as passages from the existing works are not copied verbatim or nearly verbatim, and as long as the new work does not have substantially the same structure as any of the existing works.

Use of the clean room technique puts engineers and enterprises in a legal gray area. If the owner of the original copyright or patent can demonstrate that the development of a new product was done by means of reverse engineering and is not significantly different from the existing product, a lawsuit may result. Any attempt to reverse engineer an existing product, and then create a new product based on the results of the reverse engineering process, should be undertaken only with the advice of a reputable attorney who is experienced in copyright infringement and reverse engineering issues**.**

KARPAGAM ACADEMY OF HIGHER EDUCATION

CLASS:IIM.Com(CA)

COURSE NAME: SOFTWARE MODELS AND ENGINEERING

COURSECODE:18CCP301

BATCH-2018-2020

Managing Software Projects & Design Engineering: The management spectrum, software quality, measurement and metrics. Software project estimation, decomposition techniques. Empirical estimation models (COCOMO), the Make & Buy Decision. System models: Context Models, Behavioral models, Data models, Object models. Design process, Design quality and design model. Fundamental issues in software design: Goodness of design, cohesions, coupling. Function-oriented design and object – oriented concepts. Architectural styles and patterns, Architectural Design: Unified Modeling Language (UML), User interface design. Risk Analysis and management.

MANAGEMENT SPECTRUM

Effective software project management focuses on the **four P's: people, product, process, and project**. The order is not arbitrary. The manager who forgets that software engineering work is an intensely human endeavor will never have success in project management. A manager who fails to encourage comprehensive stakeholder communication early in the evolution of a product risks building an elegant solution for the wrong problem. The manager who pays little attention to the process runs the risk of inserting competent technical methods and tools into a vacuum. The manager who embarks without a solid project plan jeopardizes the success of the project.

ThePeople

The cultivation of motivated, highly skilled software people has been discussed since the 1960s. In fact, the "people factor" is so important that the Software Engineering Institute has developed a *People Capability Maturity Model* (People-CMM), in recognition of the fact that "every organization needs to continually improve its ability to attract, develop, motivate, organize, and retain the workforce needed to accomplish its strategic business objectives". The people capability maturity model defines the following key practice areas for software people:

staffing, communication and coordination, work environment, performance management, training, compensation, competency analysis and development, career development, workgroup development, team/culture development, and others. Organizations that achieve high levels of People-CMM maturity have a higher likelihood of implementing effective software project management practices. The People-CMM is a companion to the *Software Capability Maturity Model– Integration* that guides organizations in the creation of a mature software process.

The Product

Before a project can be planned, product objectives and scope should be established, alternative solutions should be considered, and technical and management constraints should be identified. Without this information, it is impossible to define reasonable (and accurate) estimates of the cost, an effective assessment of risk, a realistic breakdown of project tasks, or a manageable project schedule that provides a meaningful indication of progress. As a software developer, you and other stakeholders must meet to define product objectives and scope. In many cases, this activity begins as part of the system engineering or business process engineering and continues as the first step in software requirements engineering. Objectives identify the overall goals for the product (from the stakeholders' points of view) without considering how these goals will be achieved. Scope identifies the primary data, functions, and behaviors that characterize the product, and more important, attempts to bound these characteristics in a quantitative manner. Once the product objectives and scope are understood, alternative solutions are considered. Although very little detail is discussed, the alternatives enable managers and practitioners to select a "best" approach, given the constraints imposed by delivery deadlines, budgetary restrictions, personnel availability, technical interfaces, and myriad other factors.

The Process

A software process provides the framework from which a comprehensive plan for software development can be established. A small number of framework activities are applicable to all software projects, regardless of their size or complexity. A number of different task sets—tasks, milestones, work products, and quality assurance points—enable the framework activities to be adapted to the characteristics of the software project and the requirements of the project team. Finally, umbrella activities—such as software quality assurance, software configuration management, and measurement—overlay the process model. Umbrella activities are independent of any one framework activity and occur throughout the process.

The Project

We conduct planned and controlled software projects for one primary reason—it is the only known way to manage complexity. And yet, software teams still struggle. In a study of 250 large software projects between 1998 and 2004, Capers Jones found that "about 25 were deemed successful in that they achieved their schedule, cost, and quality objectives. About 50 had delays or overruns below 35 percent, while about 175 experienced major delays

and overruns, or were terminated without completion." Although the success rate for present-day software projects may have improved somewhat, our project failure rate remains much higher than it should be. To avoid project failure, a software project manager and the software engineers who build the product must avoid a set of common warning signs, understand the critical success factors that lead to good project management, and develop a commonsense approach for planning, monitoring, and controlling the project.

Software quality, measurement and metrics

Software quality assurance is composed of a variety of tasks associated with two different constituencies—the software engineers who do technical work and an SQA group that has responsibility for quality assurance planning, oversight, record keeping, analysis, and reporting.

Software engineers address quality (and perform quality control activities) by applying solid technical methods and measures, conducting technical reviews, and performing well-planned software testing.

Measurements in the physical world can be categorized in two ways: direct measures (e.g., the length of a bolt) and indirect measures (e.g., the "quality" of bolts produced, measured by counting rejects). Software metrics can be categorized similarly.

Direct measures of the software process include cost and effort applied. Direct measures of the product include lines of code (LOC) produced, execution speed, memory size, and defects reported over some set period of time. Indirect measures of the product include functionality, quality, complexity, efficiency, reliability, maintainability, and many other "–abilities".

The cost and effort required to build software, the number of lines of code produced, and other direct measures are relatively easy to collect, as long as specific conventions for measurement are established in advance.

However, the quality and functionality of software or its efficiency or maintainability are more difficult to assess and can be measured only indirectly. The software metrics domain can be partitioned into process, project, and product metrics. Project metrics are then consolidated to create process metrics that are public to the software organization as a whole. But how does an organization combine metrics that come from different individuals or projects? To illustrate, consider a simple example. Individuals on two different project teams record and categorize all errors that they find during the software process. Individual measures are then combined to develop team measures. Team A found 342 errors during the software process prior to release. Team B found 184 errors. All other things being equal, which team is more effective in uncovering errors throughout the process? Because you do not know the size or complexity of the projects, you cannot answer this question. However, if the measures are normalized, it is possible to create software metrics that enable comparison to broader organizational averages.

Size-Oriented Metrics

Size-oriented software metrics are derived by normalizing quality and/or productivity measures by considering the size of the software that has been produced. If a software organization maintains simple records, a table of size-oriented measures can be created.

Function-Oriented Metrics

Function-oriented software metrics use a measure of the functionality delivered by the application as a normalization value. The most widely used function-oriented metric is the function point (FP). Computation of the function point is based on characteristics of the software's information domain and complexity.

The function point, like the LOC measure, is controversial. Proponents claim that FP is programming language independent, making it ideal for applications using conventional

and nonprocedural languages, and that it is based on data that are more likely to be known early in the evolution of a project, making FP more attractive as an estimation approach. Opponents claim that the method requires some "sleight of hand" in that computation is based on subjective rather than objective data, that counts of the information domain (and other dimensions) can be difficult to collect after the fact, and that FP has no direct physical meaning—it's just a number.

Object-Oriented Metrics

Conventional software project metrics (LOC or FP) can be used to estimate object orientedsoftware projects. However, these metrics do not provide enough granularity for the schedule and effort adjustments that are required as you iterate through an evolutionary or incremental process.

Lorenz and Kidd suggest the followingset of metrics for Object Oriented projects:

Number of scenario scripts:

A scenario script is a detailed sequence of steps that describe theinteraction between the user and the application. Each script is organized into triplets of the form

{initiator, action, participant}

where initiator is the object that requests some service (that initiates a message), action is the result of the request, and participant is the server object that satisfies the request. The number of scenario scripts is directly correlated to the size of the application and to the number of test cases that must be developed to exercise the system once it is constructed.

Number of key classes: Key classes are the "highly independent components" that are defined early in object-oriented analysis . Because key classes are central to the problem domain, the number of such classes is an

indication of the amount of effort required to develop the software and also an indicationof the potential amount of reuse to be applied during system development.

Number of support classes: Support classes are required to implement the system but are not immediately related to the problem domain. Examples might be user interface (GUI) classes, database access and manipulation classes, and computation classes. In addition, support classes can be developed for each of the key classes. Support classes are defined iteratively throughout an evolutionary process. The number of support classes is an indication of the amount of effort required to develop the software and also an indication of the potential amount of reuse to be applied during system development.

Average number of support classes per key class: In general, key classes areknown early in the project. Support classes are defined throughout. If the average number of support classes per key class were known for a given problem domain, estimating (based on total number of classes) would be greatly simplified. Lorenz and Kidd suggest that applications with a GUI have between two and three times the number of support classes as key classes. Non-GUI applications have between one and two times the number of support classes as key classes.

Number of subsystems: A subsystem is an aggregation of classes that support a function that is visible to the end user of a system. Once subsystems are identified, it is easier to lay out a reasonable schedule in which work on subsystems is partitioned among project staff.

Use-Case–Oriented Metrics: Use cases are used widely as a method for describing customer-level or business domain requirements that imply software features and functions. It would seem reasonable to use the use case as a normalization measure similar to LOC or FP.

SOFTWARE PROJECT ESTIMATION

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

1. Delay estimation until late in the project (obviously, we can achieve 100 percentaccurate estimates after the project is complete!).
2. Base estimates on similar projects that have already been completed.
3. Use relatively simple decomposition techniques to generate project cost andeffort estimates.
4. Use one or more empirical models for software cost and effort estimation.

Unfortunately, the first option, however attractive, is not practical. Cost estimatesmust be provided up-front. However, you should recognize that the longer you wait,the more you know, and the more you know, the less likely you are to make seriouserrors in your estimates.The second option can work reasonably well, if the current

project is quite similarto past efforts and other project influences (e.g., the customer, business conditions,the software engineering environment, deadlines) are roughly equivalent. Unfortunately,past experience has not always been a good indicator of future results.The remaining options are viable approaches to software project estimation.Ideally, the techniques noted for each option should be applied in tandem; each usedas a cross-check for the other.

Decomposition techniques

Software project estimation is a form of problem solving, and in most cases, the problem to be solved (i.e., developing a cost and effort estimate for a software project) is too complex to be considered in one piece. For this reason, you should decompose the problem, recharacterizing it as a set of smaller (and hopefully, more manageable) problems.But before an estimate can be made, you must understand the scope of the software to be built and generate an estimate of its "size."

Software Sizing

The accuracy of a software project estimate is predicated on a number of things:

(1) the degree to which you have properly estimated the size of the product to be built;

(2) the ability to translate the size estimate into human effort, calendar time, and dollars

(a function of the availability of reliable software metrics from past projects);

(3) the degree to which the project plan reflects the abilities of the software team; and

(4) the stability of product requirements and the environment that supports the software

engineering effort.

Putnam and Myers suggest four different approaches to the sizing problem:

• "Fuzzy logic" sizing. This approach uses the approximate reasoning techniques that are the cornerstone of fuzzy logic. To apply this approach, the planner must identify the type of application, establish its magnitude on a qualitative scale, and then refine the magnitude within the original range.

Function point sizing. The planner develops estimates of the information domain characteristics

Standard component sizing. Software is composed of a number of different "standard components" that are generic to a particular application area. For example, the standard components for an information system are subsystems, modules, screens, reports, interactive programs, batch programs, files, LOC, and object-level instructions. The project planner estimates the number of occurrences of each standard component and then uses historical project data to estimate the delivered size per standard component.

Change sizing. This approach is used when a project encompasses the use of existing software that must be modified in some way as part of a project. The planner estimates the number and type (e.g., reuse, adding code, changing code, deleting code) of modifications that must be accomplished.

Problem-Based Estimation

LOC and FP data are used in two ways during software project estimation:

(1) as estimation variables to "size" each element of the software and

(2) as baseline metrics collected from past projects and used in conjunction with estimation variables to develop cost and effort projections.

LOC and FP estimation are distinct estimation techniques. Yet both have a number of characteristics in common. You begin with a bounded statement of software scope and from this statement attempt to decompose the statement of scope into problem functions that can each be estimated individually. LOC or FP (the estimation variable) is then estimated for each function. Alternatively, you may choose another component for sizing, such as classes or objects, changes, or business processes affected.

The LOC and FP estimation techniques differ in the level of detail required for decomposition

and the target of the partitioning. When LOC is used as the estimation variable, decomposition is absolutely essential and is often taken to considerable levels of detail. The greater the degree of partitioning, the more likely reasonably accurate estimates of LOC can be developed. For FP estimates, decomposition works differently. Rather than focusing on function, each of the information domain characteristics—inputs, outputs, data files, inquiries, and external interfaces are estimated. The resultant estimates can then be used to derive an FP value that can be tied to past data and used to generate an estimate.

Regardless of the estimation variable that is used, you should begin by estimating a range of values for each function or information domain value. Using historical data or (when all else fails) intuition, estimate an optimistic, most likely, and pessimistic size value for each function or count for each information domain value. An implicit indication of the degree of uncertainty is provided when a range of values is specified. A three-point or expected value can then be computed. The expected value for the estimation variable (size) S can be computed as a weighted average of the optimistic (s_{opt}), most likely (s_m), and pessimistic (s_{pess}) estimates.

For example,

$$S = \frac{s_{opt} + 4s_m + s_{pess}}{6}$$

COCOMO - COnstructiveCOstMOdel. The original COCOMO model became one of the most widely used and discussed software cost estimation models in the industry. It has evolved intoa more comprehensive estimation model, called COCOMOII . Like its predecessor, COCOMO II is actually a hierarchy of estimation models that address the following areas:

• Application composition model. Used during the early stages of software engineering,

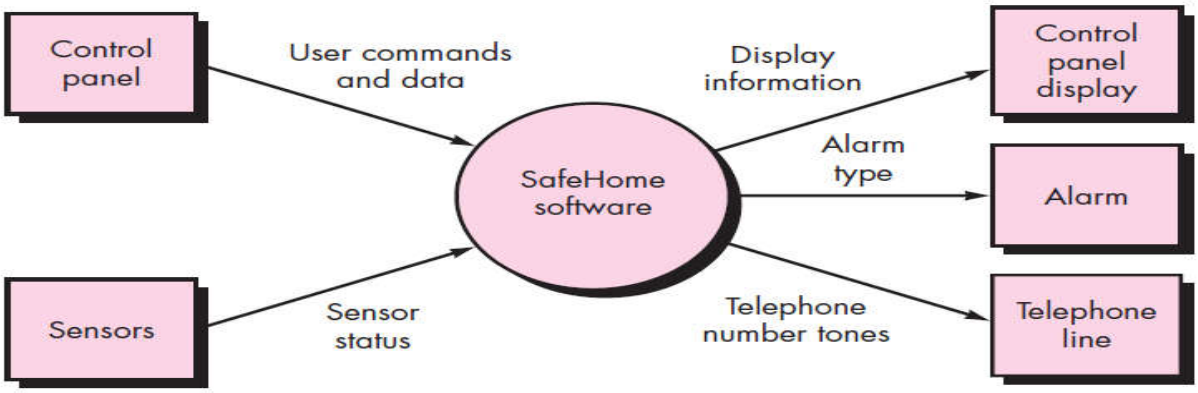
when prototyping of user interfaces, consideration of software and system interaction, assessment of performance, and evaluation of technology maturity are paramount.

• Early design stage model. Used once requirements have been stabilized and basic software architecture has been established.

• Post-architecture-stage model. Used during the construction of the software. Like all estimation models for software, the COCOMO II models require sizing information. Three different sizing options are available as part of the model hierarchy: object points, function points, and lines of source code.

Context model

The fundamental system modelor context diagram depicts the security function as a single transformation, representingthe external producers and consumers of data that flow into and out of thefunction. Figure below depicts a level 0 context model



Context-level DFD for the SafeHome security function

The behavioral model

The behavioral model indicates how software will respond to external events or stimuli. To create the model, you should perform the following steps:

1. Evaluate all use cases to fully understand the sequence of interaction within
the system.
2. Identify events that drive the interaction sequence and understand how theseevents relate to specific objects.
3. Create a sequence for each use case.
4. Build a state diagram for the system.

5. Review the behavioral model to verify accuracy and consistency.

Identifying Events with the Use Case

In general, an event occurs whenever the system and an actor exchange information.

State Representations

In the context of behavioral modeling, two different characterizations of states must be considered: (1) the state of each class as the system performs its function and

(2) the state of the system as observed from the outside as the system performs its function.

The state of a class takes on both passive and active characteristics.

A passive state is simply the current status of all of an object's attributes.

State diagrams for analysis classes.

One component of a behavioral model is a UML state diagram that represents active states for each class and the events (triggers) that cause changes between these active states. Figure below illustrates a state diagram for the Control Panel object in the SafeHome security function.

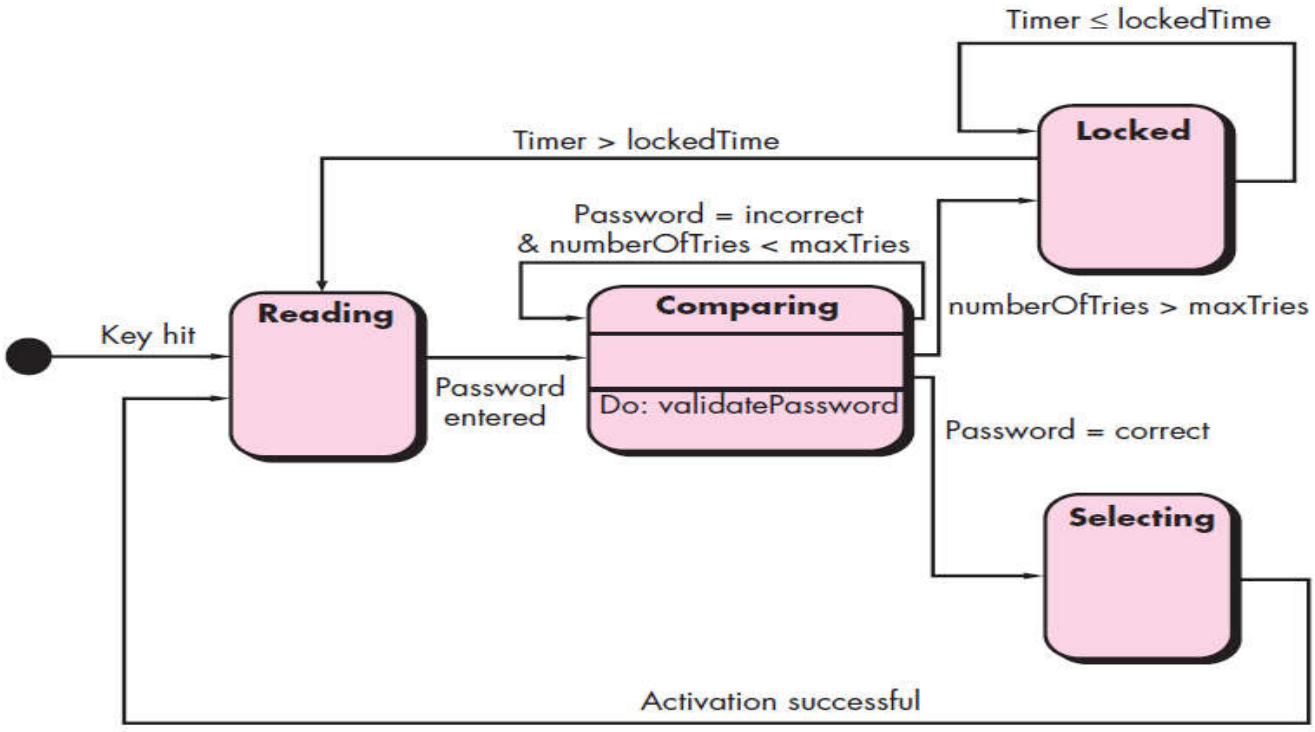


Figure : State diagram for the ControlPanel class

Sequence diagrams. The second type of behavioral representation, called a sequence diagram in UML, indicates how events cause transitions from object to object. Once events have been identified by examining a use case, the modeler creates a sequence diagram—a representation of how events cause flow from one object to another as a function of time. In essence, the sequence diagram is a shorthand version of the use case. It represents key classes and the events that cause

behavior to flow from class to class.

DATA MODEL

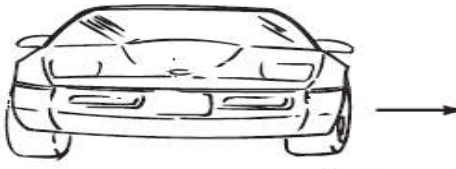
If software requirements include the need to create, extend, or interface with a database or if complex data structures must be constructed and manipulated, the software team may choose to create a data model as part of overall requirements modeling. A software engineer or analyst defines all data objects that are processed within the system, the relationships between the data objects, and other information that is pertinent to the relationships. The entity-relationship diagram (ERD) addresses these issues and represents all data objects that are entered, stored, transformed, and produced within an application.

Data Objects

A data object is a representation of composite information that must be understood by software. By composite information, I mean something that has a number of different properties or attributes. Therefore, width (a single value) would not be a valid data object, but dimensions (incorporating height, width, and depth) could be defined as an object.

Data Attributes

Data attributes define the properties of a data object and take on one of three different characteristics. They can be used to (1) name an instance of the data object, (2) describe the instance, or (3) make reference to another instance in another table. In addition, one or more of the attributes must be defined as an identifier—that is, the identifier attribute becomes a "key" when we want to find an instance of the data object. In some cases, values for the identifier(s) are unique, although this is not a requirement. Referring to the data object car, a reasonable identifier might be the ID number.



Naming attributes

Ties one data object to another, in this case, owner

Identifier

Descriptive attributes

Referential attributes

Make	Model	ID#	Body type	Color	Owner
Lexus	LS400	AB123. . .	Sedan	White	RSP
Chevy	Corvette	X456. . .	Sports	Red	CCD
BMW	750iL	XZ765. . .	Coupe	White	LJL
Ford	Taurus	Q12A45. . .	Sedan	Blue	BLF

Instance

Relationships

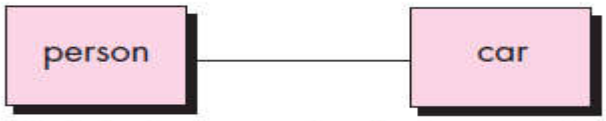
Data objects are connected to one another in different ways. Consider the two data objects, person and car. These objects can be represented using the simple notation illustrated in Figure below. A connection is established between person and car because the two objects are related. But what are the relationships? To determine the

answer, you should understand the role of people (owners, in this case) and cars within the context of the software to be built. You can establish a set of object/relationship pairs that define the relevant relationships.

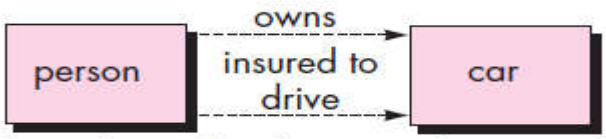
For example,

• A person owns a car.

• A person is insured to drive a car.



(a) A basic connection between data objects



(b) Relationships between data objects

Design Process

Software design is an iterative process through which requirements are translatedinto a blueprint" for constructing the software. Initially, the blueprint depicts a holisticview of software. That is, the design is represented at a high level of abstraction—a level that can be directly traced to the specific system objective and more detaileddata, functional, and behavioral requirements. As design iterations occur, subsequentrefinement leads to design representations at much lower levels of abstraction.These can still be traced to requirements, but the connection is more subtle.

Design Quality

Throughout the design process, the quality of the evolving design is assessed with a series of technical reviews. In order to evaluate the quality of a design representation, you and other members of the software team must establish technical criteria for good design.

Quality Guidelines

1. A design should exhibit an architecture that has been created using recognizable architectural styles or patterns is composed of componentsthat exhibit good design characteristics and can be implemented in an evolutionary fashion,2 thereby facilitating implementation and testing.
2. A design should be modular; that is, the software should be logically partitionedinto elements or subsystems.
3. A design should contain distinct representations of data, architecture, interfaces, and components.

4. A design should lead to data structures that are appropriate for the classes tobe implemented and are drawn from recognizable data patterns.
5. A design should lead to components that exhibit independent functionalcharacteristics.
6. A design should lead to interfaces that reduce the complexity of connectionsbetween components and with the external environment.
7. A design should be derived using a repeatable method that is driven by informationobtained during software requirements analysis.
8. A design should be represented using a notation that effectively communicatesits meaning.

Cohesion and coupling

Cohesionis an indication of the relative functional strength of a module. *Coupling* is an indication of the relative interdependence among modules. Cohesion is a natural extension of the information-hiding concept. A cohesive module performs a single task, requiring little interaction with other components in other parts of a program. Stated simply, a cohesive module should (ideally) do just one thing. Although you should always strive for high cohesion (i.e., single-mindedness), it is often necessary and advisable to have a software component perform multiple functions. However, "schizophrenic" components (modules that perform many unrelated functions) are to be avoided if a good design is to be achieved.

Coupling is an indication of interconnection among modules in a software structure.Coupling depends on the interface complexity between modules, the point atwhich entry or reference is made to a module, and what data pass across the interface.In software design, you should strive for the lowest possible coupling. Simpleconnectivity among modules results in software that is easier to understand and lessprone to a "ripple effect", caused when errors occur at one location and propagatethroughout a system.

Object Oriented Concepts

Requirements modeling (also called analysis modeling) focuses primarily on classes that are extracted directly from the statement of the problem. These entity classes typically represent things that are to be stored in a database and persist throughout the duration of the application (unless they are specifically deleted). Design refines and extends the set of entity classes. Boundary and controller classes are developed and/or refined during design. Boundary classes create the interface (e.g., interactive screen and printed reports) that the user sees and interactswith, as the software is used. Boundary classes are designed with the responsibility of managing the way entity objects are represented to users.

Controller classes are designed to manage:

(1) the creation or update of entityobjects

(2) the instantiation of boundary objects as they obtain information from entity objects,

(3) complex communication between sets of objects, and

(4) validation of data communicated between objects or between the user and the application.

The concepts discussed in the paragraphs that follow can be useful in analysis and design work.

Inheritance.

Inheritance is one of the key differentiators between conventional and object-oriented systems. A subclass Y inherits all of the attributes and operations associated with its superclass X. This means that all data structures and algorithms originally designed and implemented for X are immediately available for Y—no further work need be done. Reuse has been accomplished directly. Any change to the attributes or operations contained within a superclass is immediately inherited by all subclasses. Therefore, the class hierarchy becomes a mechanism through which changes (at high levels) can be immediately propagated through a system. It is important to note that at each level of the class hierarchy new attributes and operations may be added to those that have been inherited from higher levels in the hierarchy. In fact, whenever a new class is to be created, you have a number of options:

• The class can be designed and built from scratch. That is, inheritance is not used.

• The class hierarchy can be searched to determine if a class higher in the hierarchy contains most of the required attributes and operations. The new class inherits from the higher class and additions may then be added, as required.

• The class hierarchy can be restructured so that the required attributes and operations can be inherited by the new class.

• Characteristics of an existing class can be overridden, and different versions of attributes or operations are implemented for the new class.

Like all fundamental design concepts, inheritance can provide significant benefit for the design, but if it is used inappropriately, it can complicate a design unnecessarily and lead to error-prone software that is difficult to maintain.

Messages.

Classes must interact with one another to achieve design goals. A message stimulates some behavior to occur in the receiving object. The behavior is accomplished when an operation is executed.

Polymorphism. *Polymorphism* is a characteristic that greatly reduces the effort required to extend the design of an existing object-oriented system. To understand polymorphism, consider a conventional application that must draw four different types of graphs: line graphs, pie charts, histograms, and Kiviat diagrams. Ideally, once data are collected for a particular type of graph, the graph should draw itself.To accomplish this in a conventional application (and maintain module cohesion), it would be necessary to develop drawing modules for each type of graph.

UML

The Unified Modeling Language (UML) is “a standard language for writingsoftware blueprints. UML may be used to visualize, specify, construct, anddocument the artifacts of a software-intensive system”. In otherwords, just as building architects create blueprints to be used by a constructioncompany, software architects create UML diagrams to help software developersbuild the software.

To model classes, including their attributes, operations, and their relationships and associations with other classes, UML provides a class diagram. A class diagram provides a static or structural view of a system. It does not show the dynamic nature of the communications between the objects of the classes in the diagram. The main elements of a class diagram are boxes, which are the icons used to represent classes and interfaces. Each box is divided into horizontal parts. The top part contains the name of the class. The middle section lists the attributes of the class. An attribute refers to something that an object of that class knows or can provide all the time. Attributes are usually implemented as fields of the class, but they need not be. They could be values that the class can compute from its instance variables or values that the class can get from other objects of which it is composed.

For example, an object might always know the current time and be able to return it to you whenever you ask. Therefore, it would be appropriate to list the current time as an attribute of that class of objects. However, the object would most likely not have that time stored in one of its instance variables, because it would need to continually update that field. Instead, the object would likely compute the current time (e.g., through consultation with objects of other classes) at the moment when the time is requested. The third section of the class diagram contains the operations or behaviors of the class. An operation refers to what objects of the class can do. It is usually implemented as a method of the class.

Figure A1 presents a simple example of a Thoroughbred class that models thoroughbred horses. It has three attributes displayed—mother, father, and birthyear. The diagram also shows three operations: getCurrentAge(), getFather(), and getMother(). There may be other suppressed attributes and operations not shown in the diagram.

Figure A1 : A class diagram for a Thoroughbred class

Each attribute can have a name, a type, and a level of visibility. The type and visibility are optional. The type follows the name and is separated from the name by a colon. The visibility is indicated by a preceding –, #, ~, or +, indicating, respectively, private, protected, package, or public visibility. In Figure A1, all attributes have private visibility, as indicated by the leading minus sign (–). You can also specify that an attribute is a static or class

attribute by underlining it. Each operation can also be displayed with a level of visibility, parameters with names and types, and a return type. An abstract class or abstract method is indicated by the use of italics for the name in the class diagram. See the Horse class in Figure A2 for an example. An interface is indicated by adding the phrase "«interface»" (called a stereotype) above the name. See the OwnedObject interface in Figure A2. An interface can also be represented graphically by a hollow circle. It is worth mentioning that the icon representing a class can have other optional parts. For example, a fourth section at the bottom of the class box can be used to list the responsibilities of the class. This fourth section is not shown in any of the figures in this appendix. Class diagrams can also show relationships between classes. A class that is a subclass of another class is connected to it by an arrow with a solid line for its shaft and with a triangular hollow arrowhead. The arrow points from the subclass to the superclass. In UML, such a relationship is called a generalization. For example, in Figure A2, the Thoroughbred and QuarterHorse classes are shown to be subclasses of the Horse abstract class. An arrow with a dashed line for the arrow shaft indicates implementation of an interface. In UML, such a relationship is called a realization. For example, in Figure A2, the Horse class implements or realizes the OwnedObject interface.

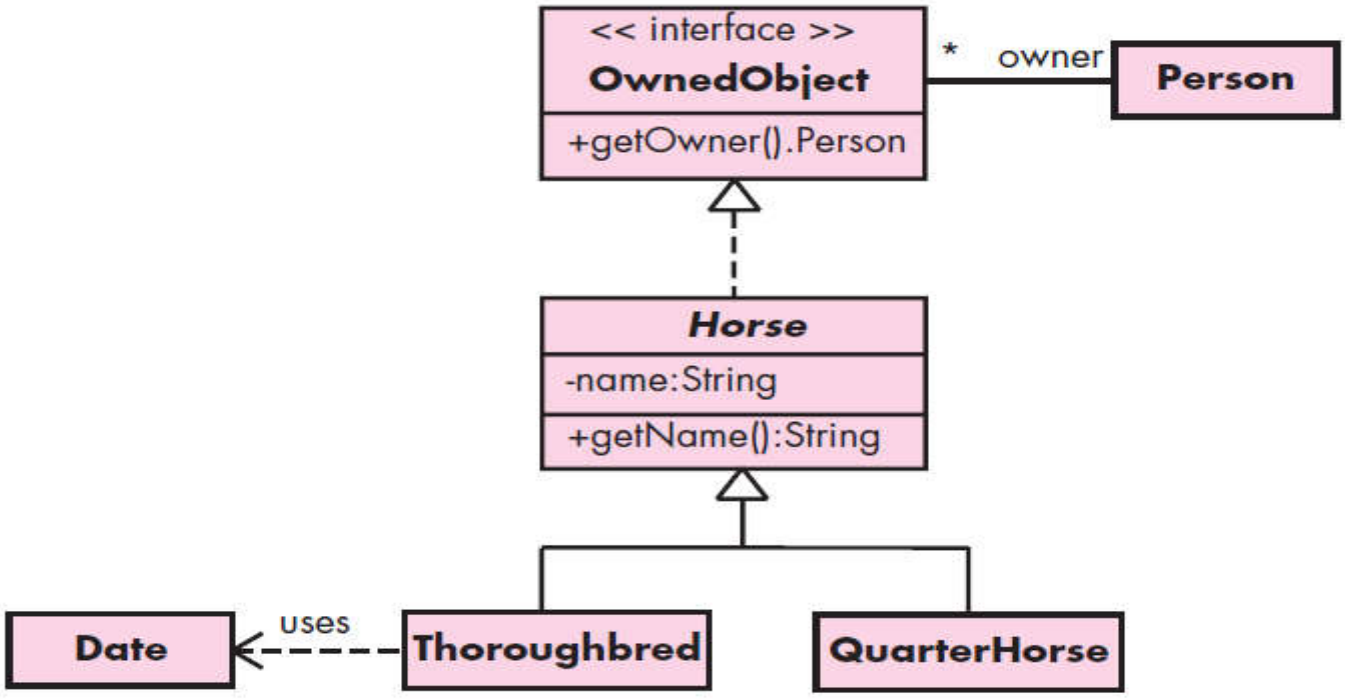


Figure A2: A class diagram regarding horses

An ***association*** between two classes means that there is a structural relationship between them. Associations are represented by solid lines. An association has many optional parts. It can be labeled, as can each of its ends, to indicate the role of each class in the association. For example, in Figure A2, there is an association between **OwnedObject**and **Person** in which the **Person** plays the role of owner. Arrows on either or both ends of an

association line indicate navigability. Also, each end of the association line can have a multiplicity value displayed. Navigability and multiplicity are explained in more detail later in this section. An association might also connect a class with itself, using a loop. Such an association indicates the connection of an object of the class with other objects of the same class. An association with an arrow at one end indicates one-way navigability. The arrow means that from one class you can easily access the second associated class to which the association points, but from the second class, you cannot necessarily easily access the first class. Another way to think about this is that the firs class is aware of the second class, but the second class object is not necessarily directly aware of the first class. An association with no arrows usually indicates a two-way association, which is what was intended in Figure A2, but it could also just mean that the navigability is not important and so was left off.

It should be noted that an attribute of a class is very much the same thing as an association of the class with the class type of the attribute. That is, to indicate that a class has a property called "name" of type String, one could display that property as an attribute, as in the **Horse** class in Figure A2. Alternatively, one could create a one-way association from the **Horse** class to the **String** class with the role of the **String** class being "name." The attribute approach is better for primitive data types, whereas the association approach is often better if the property's class plays a major role in the design, in which case it is valuable to have a class box for that type. A *dependency* relationship represents another connection between classes and is indicated by a dashed line (with optional arrows at the ends and with optional labels). One class depends on another if changes to the second class might require changes to the first class. An association from one class to another automatically indicates a dependency. No dashed line is needed between classes if there is already an association between them.

However, for a transient relationship (i.e., a class that does not maintain any long-term connection to another class but does use that class occasionally) we should draw a dashed line from the first class to the second. For example, in Figure A2, the **Thoroughbred** class uses the **Date** class whenever its *getCurrentAge()* method is invoked, and so the dependency is labeled "uses." The *multiplicity* of one end of an association means the number of objects of that class associated with the other class. A multiplicity is specified by a nonnegative integer or by a range of integers. A multiplicity specified by "0..1" means that there are 0 or 1 objects on that end of the association. For example, each person in the world has either a Social Security number or no such number (especially if they are not U.S. citizens), and so a multiplicity of 0..1 could be used in an association between a**Person** class and a **SocialSecurityNumber**class in a class diagram. A multiplicityspecified by "1..*" means one or more, and a multiplicity specified by "0..*" or just "*"means zero or more. An * was used as the multiplicity on the **OwnedObject**end ofthe association with class **Person** in Figure A2 because a **Person** can own zero ormore objects..

If one end of an association has multiplicity greater than 1, then the objects of the class referred to at that end of the association are probably stored in a collection, such as a set or ordered list. One could also include that

collection class itself in the UML diagram, but such a class is usually left out and is implicitly assumed to be there due to the multiplicity of the association.

An***aggregation***is a special kind of association indicated by a hollow diamond on one end of the icon. It indicates a "whole/part" relationship, in that the class to which the arrow points is considered a "part" of the class at the diamond end of the association.

A ***composition***is an aggregation indicating strong ownership of the parts. In a composition, the parts live and die with the owner because they have no role in the software system independent of the owner.

UML *use-case diagram* help you determine the functionality and features of the software from the user's perspective. To give you a feeling for how use cases and use-case diagrams work, we will create some for a software application for managing digital music files, similar to Apple's iTunes software.

Some of the things the software might do include:

• Download an MP3 music file and store it in the application's library.

• Capture streaming music and store it in the application's library.

• Manage the application's library (e.g., delete songs or organize them in playlists).

• Burn a list of the songs in the library onto a CD.

• Load a list of the songs in the library onto an iPod or MP3 player.

• Convert a song from MP3 format to AAC format and vice versa.

A use-case diagram for the digital music application is shown in Figure A3.

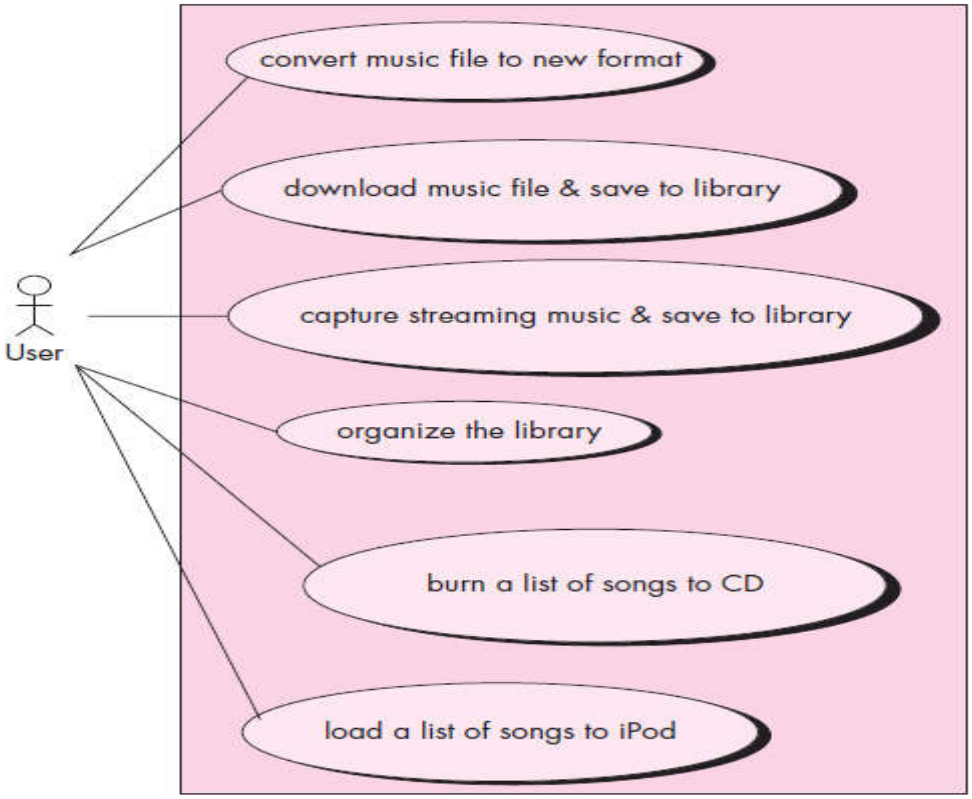


Figure A3: Use case diagram for music system

RISK ANALYSIS

Software development is activity that uses a variety of technological advancements and requires high levels of knowledge. Because of these and other factors, every software development project contains elements of uncertainty. This is known as project risk. The success of a software development project depends quite heavily on the amount of risk that corresponds to each project activity. As a project manager, it's not enough to merely be aware of the risks. To achieve a successful outcome, project leadership must identify, assess, prioritize, and manage all of the major risks. Risk is the possibility of suffering loss, and total risk exposure to a specific project will account for both the *probability* and the *size* of the potential loss.

Risk management

Risk management means risk containment and mitigation. First, you've got to identify and plan. Then be ready to act when a risk arises, drawing upon the experience and knowledge of the entire team to minimize the impact to the project.

Risk management includes the following tasks:

- ***Identify*** risks and their triggers
- ***Classify*** and prioritize all risks
- Craft a ***plan*** that links each risk to a mitigation
- ***Monitor*** for risk triggers during the project
- Implement the ***mitigating action*** if any risk materializes
- ***Communicate*** risk status throughout project

Identify and Classify Risks

Most software engineering projects are inherently risky because of the variety potential problems that might arise. Experience from other software engineering projects can help managers classify risk. The importance here is not the elegance or range of classification, but rather to precisely identify and describe all of the real threats to project success. A simple but effective classification scheme is to arrange risks according to the areas of impact.

Five Types of Risk In Software Project Management

For most software development projects, we can define five main risk impact areas:

- New, unproven technologies
- User and functional requirements
- Application and system architecture
- Performance
- Organizational

New, unproven technologies. The majority of software projects entail the use of new technologies. Ever-changing tools, techniques, protocols, standards, and development systems increase the probability that technology risks will arise in virtually any substantial software engineering effort. Training and knowledge are of critical importance, and the improper use of new technology most often leads directly to project failure.

User and functional requirements. Software requirements capture all user needs with respect to the software system features, functions, and quality of service. Too often, the process of requirements definition is lengthy, tedious, and complex. Moreover, requirements usually change with discovery, prototyping, and integration activities. Change in elemental requirements will likely propagate throughout the entire project, and modifications to user requirements might not translate to functional requirements. These disruptions often lead to one or more critical failures of a poorly-planned software development project.

Application and system architecture. Taking the wrong direction with a platform, component, or architecture can have disastrous consequences. As with the technological risks, it is vital that the team includes experts who understand the architecture and have the capability to make sound design choices.

Performance. It's important to ensure that any risk management plan encompasses user and partner expectations on performance. Consideration must be given to benchmarks and threshold testing throughout the project to ensure that the work products are moving in the right direction.

Organizational. Organizational problems may have adverse effects on project outcomes. Project management must plan for efficient execution of the project, and find a balance between the needs of the development team and the expectations of the customers. Of course, adequate staffing includes choosing team members with skill sets that are a good match with the project.

Risk Management Plan

After cataloging all of the risks according to type, the software development project manager should craft a risk management plan. As part of a larger, comprehensive project plan, the risk management plan outlines the response that will be taken for each risk—if it materializes.

Monitor and Mitigate

To be effective, software risk monitoring has to be integral with most project activities. Essentially, this means frequent checking during project meetings and critical events.

Monitoring includes:

- Publish project status reports and include risk management issues
- Revise risk plans according to any major changes in project schedule
- Review and reprioritize risks, eliminating those with lowest probability
- Brainstorm on potentially new risks after changes to project schedule or scope

When a risk occurs, the corresponding mitigation response should be taken from the risk management plan.

Mitigating options include:

- **Accept:** Acknowledge that a risk is impacting the project. Make an explicit decision to accept the risk without any changes to the project. Project management approval is mandatory here.
- **Avoid:** Adjust project scope, schedule, or constraints to minimize the effects of the risk.
- **Control:** Take action to minimize the impact or reduce the intensification of the risk.
- **Transfer:** Implement an organizational shift in accountability, responsibility, or authority to other stakeholders that will accept the risk.
- **Continue Monitoring:** Often suitable for low-impact risks, monitor the project environment for potentially increasing impact of the risk.

Communicate

Throughout the project, it's vital to ensure effective communication among all stakeholders, managers, developers, QA—especially marketing and customer representatives. Sharing information and getting feedback about risks will greatly increase the probability of project success.

UNIT-III

SYLLABUS

KARPAGAM ACADEMY OF HIGHER EDUCATION

CLASS:IIM.Com(CA)

COURSE NAME: SOFTWARE MODELS AND ENGINEERING

COURSECODE:18CCP301

BATCH-2018-2020

S/W Requirements, S/W Metrices& Testing Strategies: S/W Requirements : Functional and non-functional requirements, User requirements, System requirements.SRA& SRS. S/W Metrices: Process Metrices, Project Metrices& Product Metrices. Testing Strategies : A strategic approach to software testing, Testing fundamentals, Test Case Design. Types Of Testing: Black-Box Testing, White-Box Testing, Validation testing, System testing, the art of Debugging. Code walkthrough and reviews. Software Quality, Metrics for Analysis Model, Metrics for Design Model, Metrics for source code, Metrics for testing, Metrics for maintenance.

Requirements engineering (RE) is the process of establishing the services that the customer requires from a system and the constraints under which it operates and is developed. The **requirements** themselves are the descriptions of the system services and constraints that are generated during the requirements engineering process. Requirements may range from a high-level abstract statement of a service or of a system constraint to a detailed mathematical functional specification. As much as possible, requirements should describe **what** the system should do, but **not how** it should do it.

A functional requirement describes *what* **a software system should do, while non-functional requirements place constraints on** *how* **the system will do so.**

In software engineering, a functional requirement defines a system or its component. It describes the functions a software must perform. A function is nothing but inputs, its behavior, and outputs. It can be a calculation, data manipulation, business process, user interaction, or any other specific functionality which defines what function a system is likely to perform.

Functional software requirements help you to capture the intended behavior of the system. This behavior may be expressed as functions, services or tasks or which system is required to perform.

Functional requirements

The functional requirements for a system describe what the system should do. These requirements depend on the type of software being developed, the expected users of the software, and the general approach taken by the organization when writing requirements. When expressed as user requirements, functional requirements are usually described in an abstract way that can be understood by system users. However, more specific functional system requirements describe the system functions, its inputs and outputs, exceptions, etc., in detail. Functional system requirements vary from general requirements covering what the system should do to very specific requirements reflecting local ways of working or an organization's existing systems.

For example, here are examples of functional requirements for the MHC-PMS system, used to maintain information about patients receiving treatment for mental health problems:

1. A user shall be able to search the appointments lists for all clinics.

2. The system shall generate each day, for each clinic, a list of patients who are expected to attend appointments that day.

3. Each staff member using the system shall be uniquely identified by his or her eight-digit employee number.

These functional user requirements define specific facilities to be provided by the system. These have been taken from the user requirements document and they show that functional requirements may be written at different levels of detail (contrast requirements 1 and 3). Imprecision in the requirements specification is the cause of many software engineering problems. It is natural for a system developer to interpret an ambiguous requirement in a way that simplifies its implementation. Often, however, this is not what the customer wants. New requirements have to be established and changes made to the system. Of course, this delays system delivery and increases costs. For example, the first example requirement for the MHC-PMS states that a user shall be able to search the appointments lists for all clinics. The rationale for this requirement is that patients with mental health problems are sometimes confused. They may have an appointment at one clinic but actually go to a different clinic. If they have an appointment, they will be recorded as having attended, irrespective of the clinic. The medical staff member specifying this may expect 'search' to mean that, given a patient name, the system looks for that name in all appointments at all clinics. However, this is not explicit in the requirement. System developers may interpret the requirement in a different way and may implement a search so that the user has to choose a clinic then carry out the search. This obviously will involve more user input and so take longer. In principle, the functional requirements specification of a system should be both complete and consistent. Completeness means that all services required by the user should be defined. Consistency means that requirements should not have contradictory definitions. In practice, for large, complex systems, it is practically impossible to achieve

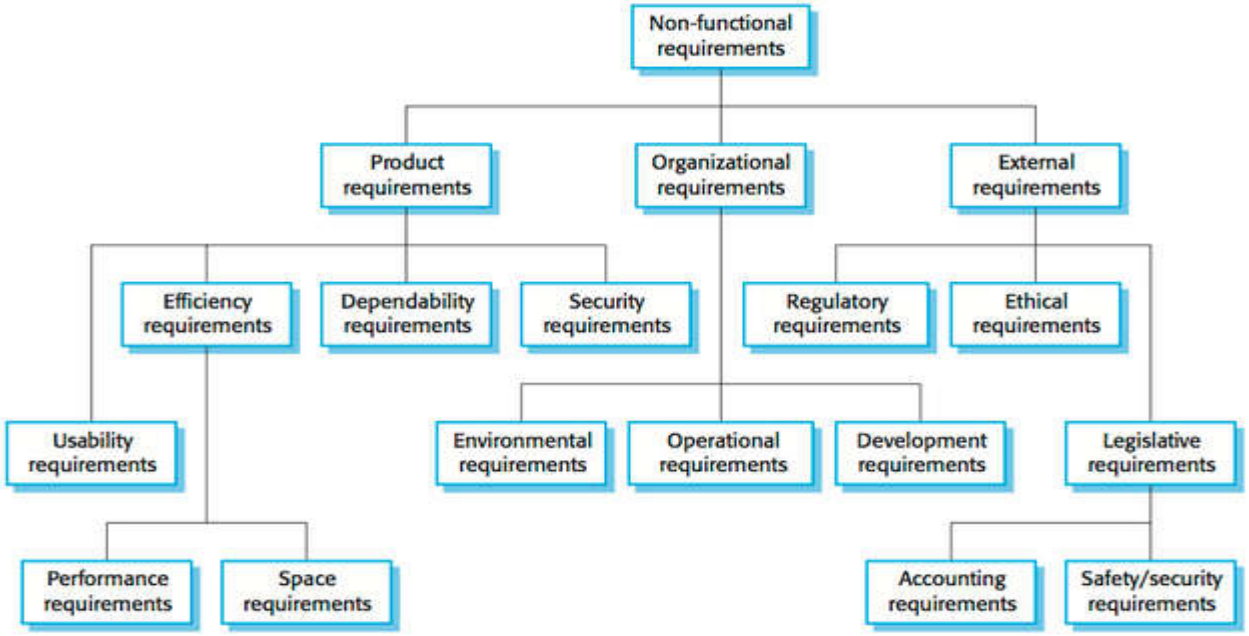
requirements consistency and completeness. One reason for this is that it is easy to make mistakes and omissions when writing specifications for complex systems. Another reason is that there are many stakeholders in a large system. A stakeholder is a person or role that is affected by the system in some way. Stakeholders have different—and often inconsistent—needs. These inconsistencies may not be obvious when the requirements are first specified, so inconsistent requirements are included in the specification. The problems may only emerge after deeper analysis or after the system has been delivered to the customer.

Non Functional Requirements

Non-functional requirements, as the name suggests, are requirements that are not directly concerned with the specific services delivered by the system to its users. They may relate to emergent system properties such as reliability, response time, and store occupancy. Alternatively, they may define constraints on the system implementation such as the capabilities of I/O devices or the data representations used in interfaces with other systems. Non-functional requirements, such as performance, security, or availability, usually specify or constrain characteristics of the system as a whole. Non-functional requirements are often more critical than individual functional requirements. System users can usually find ways to work around a system function that doesn't really meet their needs. However, failing to meet a non-functional requirement can mean that the whole system is unusable. For example, if an aircraft system does not meet its reliability requirements, it will not be certified as safe for operation; if an embedded control system fails to meet its performance requirements, the control functions will not operate correctly. Although it is often possible to identify which system components implement specific functional requirements (e.g., there may be formatting components that implement reporting requirements), it is often more difficult to relate components to non-functional requirements. The implementation of these requirements may be diffused throughout the system. There are two reasons for this:

1. Non-functional requirements may affect the overall architecture of a system rather than the individual components. For example, to ensure that performance requirements are met, you may have to organize the system to minimize communications between components.

2. A single non-functional requirement, such as a security requirement, may generate a number of related functional requirements that define new system services that are required. In addition, it may also generate requirements that restrict existing requirements.



Types of non-functional requirement

Three classes of non-functional requirements:

1. Product requirements

Requirements which specify that the delivered product must behave in a particular way e.g. execution speed, reliability, etc.

2. Organizational requirements

Requirements which are a consequence of organizational policies and procedures e.g. process standards used, implementation requirements, etc.

3. External requirements

Requirements which arise from factors which are external to the system and its development process e.g. interoperability requirements, legislative requirements, etc.

Non-functional requirements may be very difficult to state precisely and imprecise requirements may be difficult to verify. If they are stated as a **goal** (a general intention of the user such as ease of use), they should be rewritten as a **verifiable** non-functional requirement (a statement using some quantifiable metric that can be objectively tested). Goals are helpful to developers as they convey the intentions of the system users.

User requirements

High-level abstract requirements written as statements, in a natural language plus diagrams, of what services the system is expected to provide to system users and the constraints under which it must operate. The **user**

requirement(s) document (URD) or user requirement(s) specification (URS) is a document usually used in software engineering that specifies what the user expects the software to be able to do. Once the required information is completely gathered it is documented in a URD, which is meant to spell out exactly what the software must do and becomes part of the contractual agreement. A customer cannot demand features not in the URD, whilst the developer cannot claim the product is ready if it does not meet an item of the URD. The URD can be used as a guide to planning cost, timetables, milestones, testing, etc. The explicit nature of the URD allows customers to show it to various stakeholders to make sure all necessary features are described. Formulating a URD requires negotiation to determine what is technically and economically feasible. Preparing a URD is one of those skills that lies between a science and an art, requiring both software technical skills and interpersonal skills.

The user requirements for a system should describe the functional and nonfunctional requirements so that they are understandable by system users who don't have detailed technical knowledge. Ideally, they should specify only the external behavior of the system. The requirements document should not include details of the system architecture or design. Consequently, if you are writing user requirements, you should not use software jargon, structured notations, or formal notations. You should write user requirements in natural language, with simple tables, forms, and intuitive diagrams.

User requirements are almost always written in natural language supplemented by appropriate diagrams and tables in the requirements document. System requirements may also be written in natural language but other notations based on forms, graphical system models, or mathematical system models can also be used. Graphical models are most useful when you need to show how a state changes or when you need to describe a sequence of actions. UML sequence charts and state charts show the sequence of actions that occur in response to a certain message or event. Formal mathematical specifications are sometimes used to describe the requirements for safety- or security-critical systems, but are rarely used in other circumstances.

System requirements

System requirements are expanded versions of the user requirements that are used by software engineers as the starting point for the system design. They add detail and explain how the user requirements should be provided by the system. They may be used as part of the contract for the implementation of the system and should therefore be a complete and detailed specification of the whole system. Ideally, the system requirements should simply describe the external behavior of the system and its operational constraints. They should not be concerned with how the system should be designed or implemented. However, at the level of detail required to completely specify a complex software system, it is practically impossible to exclude all design information. There are several reasons for this:

1. You may have to design an initial architecture of the system to help structure the requirements specification. The system requirements are organized according to the different sub-systems that make up the system.
2. In most cases, systems must interoperate with existing systems, which constrain the design and impose requirements on the new system.
3. The use of a specific architecture to satisfy non-functional requirements may be necessary. An external regulator who needs to certify that the system is safe may specify that an already certified architectural design be used.

Software requirements specification (SRS)

It is a document that describes what the software will do and how it will be expected to perform. An SRS describes the functionality the product needs to fulfill all stakeholders (business, users) needs. A software requirements specification (SRS) is a document that captures complete description about how the system is expected to perform. It is usually signed off at the end of requirements engineering phase.

A software requirements specification (SRS) is a document that is created when a detailed description of all aspects of the software to be built must be specified before the project is to commence. It is important to note that a formal SRS is not always written. In fact, there are many instances in which effort expended on an SRS might be better spent in other software engineering activities. However, when software is to be developed by a third party, when a lack of specification would create severe business issues, or when a system is extremely complex or business critical, an SRS may be justified.

Quality Characteristics of a good SRS

Correctness:

User review is used to ensure the correctness of requirements stated in the SRS. SRS is said to be correct if it covers all the requirements that are actually expected from the system.

Completeness:

Completeness of SRS indicates every sense of completion including the numbering of all the pages, resolving the to be determined parts to as much extent as possible as well as covering all the functional and non-functional requirements properly.

Consistency:

Requirements in SRS are said to be consistent if there are no conflicts between any set of requirements. Examples

of conflict include differences in terminologies used at separate places, logical conflicts like time period of report generation, etc.

 Unambiguousness: An SRS is said to be unambiguous if all the requirements stated have only 1 interpretation. Some of the ways to prevent unambiguousness include the use of modelling techniques like ER diagrams, proper reviews and buddy checks, etc.

 Ranking for importance and stability: There should a criterion to classify the requirements as less or more important or more specifically as desirable or essential. An identifier mark can be used with every requirement to indicate its rank or stability.

 Modifiability: SRS should be made as modifiable as possible and should be capable of easily accepting changes to the system to some extent. Modifications should be properly indexed and cross-referenced.

 Verifiability: An SRS is verifiable if there exists a specific technique to quantifiably measure the extent to which every requirement is met by the system. For example, a requirement stating that the system must be user-friendly is not verifiable and listing such requirements should be avoided.

 Traceability:
One should be able to trace a requirement to a design component and then to a code segment in the program. Similarly, one should be able to trace a requirement to the corresponding test cases.

 Design Independence:
There should be an option to choose from multiple design alternatives for the final system. More specifically, the SRS should not include any implementation details.

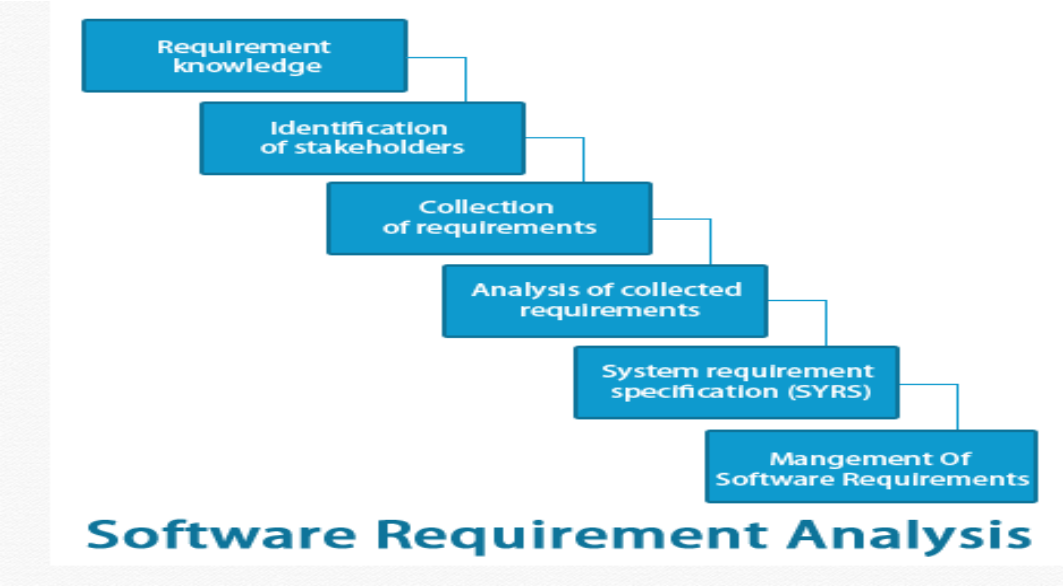
 Testability:
An SRS should be written in such a way that it is easy to generate test cases and test plans from the document.

 Understandable by the customer:
An end user maybe an expert in his/her specific domain but might not be an expert in computer science. Hence, the use of formal notations and symbols should be avoided to as much extent as possible. The language should be kept easy and clear.

 Right level of abstraction:
If the SRS is written for the requirements phase, the details should be explained explicitly. Whereas, for a feasibility study, fewer details can be used. Hence, the level of abstraction varies according to the purpose of the

Software Requirement Analysis(SRA)

Software requirement is a functional or non-functional need to be implemented in the system. Functional means providing particular service to the user. For example, in context to banking application the functional requirement will be when customer selects "View Balance" they must be able to look at their latest account balance. Software requirement can also be a non-functional, it can be a performance requirement. For example, a non-functional requirement is where every page of the system should be visible to the users within 5 seconds.



Necessity Of Requirement Analysis

According to statistics major reason of failure of software is that it does not meet with the requirement of the user. Requirement analysis involves the task that determines the needs of the software, which mainly includes complaints and needs of various clients/stakeholders.

Software Requirement Analysis Process

The steps for effective capturing on present requirements of users are:

- Requirement Knowledge:

It is very necessary to know about the requirements of the users before starting any project. Working on the present requirements of the users will be helpful in gaining popularity of your project.

- Identification of Stakeholders:

Stakeholders includes customers, end-users, system administrators etc. identifying the correct stakeholder is second step and is one of the most important step in all. Identifying the correct stakeholders help to properly analyze and create a road map for gathering requirements.

- Collection of Requirements:

After identifying stakeholders one has to collect requirements for them. Based on the nature and aim of the project there can be many kinds of stakeholders. Interacting with stakeholder groups can be in person interviews, focus groups, market study, surveys and secondary research.

- Analysis of Collected Requirements:

Once the data is gathered structured analysis must be done of the data to make models. Data are analysed on the basis of various parameters depending on the goals of the software. These include animation, automated reasoning, knowledge based critiquing, consistency checking, analogical and case based reasoning.

- System requirement Specification (SYRS):

Once the data is analyzed they are put together in the form of system requirement specification document (SYRS) or system requirement specification (SRS). It acts as a blueprint for the designing team to make the project. It serves as a technical collection of all the requirements of stake holders which includes user requirements, system requirements, user interface and operational requirements.

- Management Of Software Requirements:

The last step of this analysis process is correcting and validating all elements of requirement specifications document. Errors can be corrected at this stage. Minor changes can also be done according to the requirement of the software user.

Code Walkthrough is a form of peer review in which a programmer leads the review process and the other team members ask questions and spot possible errors against development standards and other issues.

- The meeting is usually led by the author of the document under review and attended by other members of the team.
- Review sessions may be formal or informal.
- Before the walkthrough meeting, the preparation by reviewers and then a review report with a list of findings.
- The scribe, who is not the author, marks the minutes of meeting and note down all the defects/issues so that it can be tracked to closure.
- The main purpose of walkthrough is to enable learning about the content of the document under review to help team members gain an understanding of the content of the document and also to find defects.

UNIT IV

SYLLABUS

KARPAGAM ACADEMY OF HIGHER EDUCATION

CLASS:IIM.Com(CA)

COURSE NAME: SOFTWARE MODELS AND ENGINEERING

COURSECODE:18CCP301

BATCH-2018-2020

Testing Plan and Maintenance: Snooping for information, Coping with complexity through teaming, Testing plan focus areas, Testing for recoverability, Planning for troubles, Preparing for the tests: Software Reuse, Developing good test programs , Data corruption, Tools, Test Execution ,Testing with a virtual computer, Simulation and Prototypes, Managing the Test, Customer's role in testing, Software maintenance issues and techniques. Software reuse. Client-Server software development.

Snooping for information

Snooping, in a security context, is unauthorized access to another person's or company's data. The practice is similar to eavesdropping but is not necessarily limited to gaining access to data during its transmission. Snooping can include casual observance of an e-mail that appears on another's computer screen or watching what someone else is typing. More sophisticated snooping uses software programs to remotely monitor activity on a computer or network device.

Malicious hackerkeyloggers to monitor keystrokes, capture passwords and login information, and to intercept e-mail and other private communications and data transmissions. Corporations sometimes snoop on employees legitimately to monitor their use of business computers and track Internet usage; governments may snoop on individuals to collect information and avert crime and terrorism.

Test plan focus area

A **TEST PLAN** is a document describing software testing scope and activities. It is the basis for formally testing any software/product in a project.

- **test plan**: A document describing the scope, approach, resources and schedule of intended test activities. It identifies amongst others test items, the features to be tested, the testing tasks, who will do each task, degree of tester independence, the test environment, the test design techniques and entry and exit criteria to be used, and the rationale for their choice,and any risks requiring contingency planning. It is a record of the test planning process.
- **master test plan**: A test plan that typically addresses multiple test levels.
- **phase test plan**: A test plan that typically addresses one test phase.

Test Plan Types

One can have the following types of test plans:

- **Master Test Plan:** A single high-level test plan for a project/product that unifies all other test plans.
- **Testing Level Specific Test Plans:**Plans for each level of testing.
    - Unit Test Plan
    - Integration Test Plan
    - System Test Plan
    - Acceptance Test Plan
- **Testing Type Specific Test Plans:** Plans for major types of testing like Performance Test Plan and Security Test Plan.

Recovery testing in software testing

Recovery testing is a type of non-functional **testing** technique performed in order to determine how quickly the system can **recover** after it has gone through system crash or hardware failure. **Recovery testing** is the forced failure of the **software** to verify if the **recovery** is successful.It involves reverting to a point where the integrity of the system was known and then reprocessing transactions up to the point of failure.

The purpose of recovery testing is to verify the system's ability to recover from varying points of failure.

The time taken to recover depends upon:

- The number of restart points
- A volume of the applications
- Training and skills of people conducting recovery activities and tools available for recovery.

When there are a number of failures then instead of taking care of all failures, the recovery testing should be done in a structured fashion which means recovery testing should be carried out for one segment and then another.

It is done by professional testers. Before recovery testing, adequate backup data is kept in secure locations. This is done to ensure that the operation can be continued even after a disaster.

Life Cycle of Recovery Process

The life cycle of the recovery process can be classified into the following five steps:

1. Normal operation
2. Disaster occurrence
3. Disruption and failure of the operation
4. Disaster clearance through the recovery process
5. Reconstruction of all processes and information to bring the whole system to move to normal operation

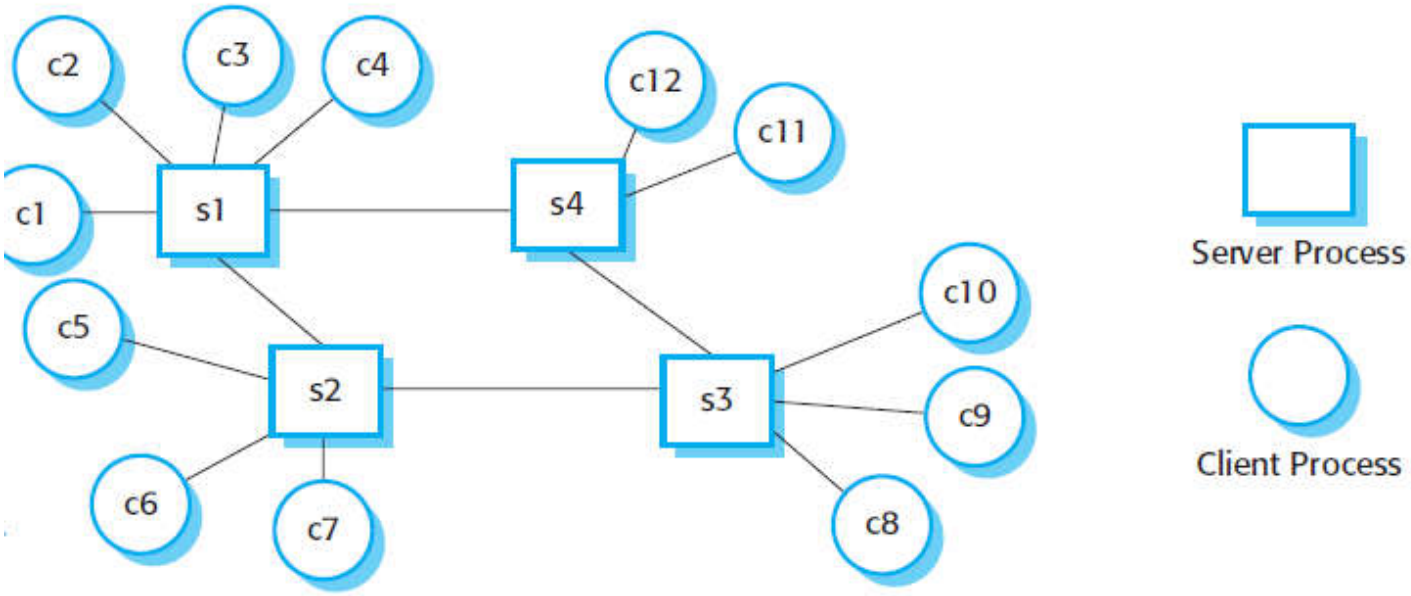
1. A system consisting of hardware, software, and firmware integrated to achieve a common goal is made operational for carrying out a well-defined and stated goal. The system is called to perform the normal operation to carry out the designed job without any disruption within a stipulated period of time.
2. A disruption may occur due to malfunction of the software, due to various reasons like input initiated malfunction, software crashing due to hardware failure, damaged due to fire, theft, and strike.
3. Disruption phase is a most painful phase which leads to business losses, relation break, opportunity losses, man-hour losses and invariably financial and goodwill losses. Every sensible agency should have a plan for disaster recovery to enable the disruption phase to be minimal.
4. If a backup plan and risk mitigation processes are at the right place before encountering disaster and disruption, then recovery can be done without much loss of time, effort and energy. A designated individual, along with his team with the assigned role of each of these persons should be defined to fix the responsibility and help the organization to save from long disruption period.
5. Reconstruction may involve multiple sessions of operation to rebuild all folders along with configuration files. There should be proper documentation and process of reconstruction for correct recovery.

While performing recovery testing following things should be considered.

- We must create a test bed as close to actual conditions of deployment as possible. Changes in interfacing, protocol, firmware, hardware, and software should be as close to the actual condition as possible if not the same condition.
- Through exhaustive testing may be time-consuming and a costly affair, identical configuration, and complete check should be performed.
- If possible, testing should be performed on the hardware we are finally going to restore. This is especially true if we are restoring to a different machine than the one that created the backup.
- Some backup systems expect the hard drive to be exactly the same size as the one the backup was taken from.
- Obsolescence should be managed as drive technology is advancing at a fast pace, and old drive may not be compatible with the new one. One way to handle the problem is to restore to a virtual machine. Virtualization software vendors like VMware Inc. can configure virtual machines to mimic existing hardware, including disk sizes and other configurations.
- Online backup systems are not an exception for testing. Most online backup service providers protect us from being directly exposed to media problems by the way they use fault-tolerant storage systems.
- While online backup systems are extremely reliable, we must test the restore side of the system to make sure there are no problems with the retrieval functionality, security or encryption.

Client server softwaredevelopment

If protection of data is a critical requirement, then a client–server architecture should be used, with the protection mechanisms built into the server. However, if the protection is compromised, then the losses associated with an attack are likely to be high, as are the costs of recovery (e.g., all user credentials may have to be reissued). The system is vulnerable to denial of service attacks, which overload the server and make it impossible for anyone to access the system database.



Client server interaction

client–server model

This is a multi-user, web-based system for providing a film and photograph library. In this system, several servers manage and display the different types of media. Video frames need to be transmitted quickly and in synchrony but at relatively low resolution. They may be compressed in a store, so the video server can handle video compression and decompression in different formats. Still pictures, however, must be maintained at a high resolution, so it is appropriate to maintain them on a separate server. The catalog must be able to deal with a variety of queries and provide links into the web information system that includes data about the film and video clips, and an e-commerce system that supports the sale of photographs, film, and video clips.

UNIT-V

SYLLABUS

KARPAGAM ACADEMY OF HIGHER EDUCATION

CLASS:IIM.Com(CA)

COURSE NAME: SOFTWARE MODELS AND ENGINEERING

COURSECODE:18CCP301

BATCH-2018-2020

Software Reengineering and Project Management: Software Reengineering, Reverse Engineering & Forward Engineering, Life Cycle Phases and Process artifacts, Restructuring. Model based software architectures, Software process and Iteration workflows, Major and Minor milestones, Periodic status assessments, Process Planning, Project Control and process instrumentation: Seven core metrics, management indicators, quality indicators, life-cycle expectations, CCPDS-R Case Study and Future Software Project Management Practices.

Techniques for Maintenance

To perform software maintenance effectively, various techniques are used. These include software configuration management, impact analysis, and software rejuvenation, all of which help in maintaining a system and thus, improve the quality of the existing system.

Software Configuration Management

Software configuration management can be used effectively while maintaining a system as it keeps track of changes and their effects on the system components. Many changes occur when the software is delivered to the users such as failure or users' request for enhancement in the software. For this, configuration control board (CCB) oversees the entire change process. Note that the representatives of CCB along with the users and developers manage changes collectively. These changes are managed in the following steps.

1. When the user encounters a problem such as failure report, he requests for change on a formal change request form. The problem can also be an enhancement to a function, variation in the older function, or deleting an existing function. The procedure for request of change remains the same. The change reqest form should include information about how the system works, nature of the problem, and how the new (expected) system should work.
2. The request for change is reported to CCB.
3. The representative of CCB meets the user to discuss the problem (That is, to determine that the request is for failure report or for enhancement).

4. If the user requests for a reported failure, the CCB discusses the source of the problem. If the requested change is an enhancement, the CCB discusses the parts or the components that will be affected by the change. In both the cases, developers describe the scope of changes and the expected time to implement them.
5. The developers determine the source of the problem or the components which will be affected when the changes will be implemented. For this, they use a test copy instead of the operational system and implement the requested changes to see whether it (test copy) performs according to the requested changes.
6. Finally, after the changes have been made, all the relevant documentation is updated according to the requested change.
7. The developers then record all the changes made to the operational system in a change report to keep track of the next release or version of the software system.

Version control implies the process by which the contents of the software, hardware, or documentation are revised. It tracks and manages the progress of files and directories within a project. This process is required when one or more components of a software system are changed (for example, Microsoft has introduced MSN Messenger 7.0, which is an upgraded version of MSN Messenger 6.2). Software maintenance manages the versions, that is, the older version (present software) and the new version (when the software is modified). Note that the software configuration management manages how the versions differ, who made the changes, and why they were made.

The component (existing version) is assigned an identification number. When the version (current) is revised, a revision number is allotted to each resulting changed component. The records such as name of the component, date and time, version status, and account of all changes are managed. This helps the software configuration management to identify the current version and the revised number of the operational system.

Impact Analysis

Impact analysis is used to evaluate the overall effect of the requested change. This includes identifying the components that will be affected with the change, the extent to which each of the components will be affected, and the consequences of change on the estimated effort and schedule. There are various advantages of performing impact analysis, which are listed below.

1. It is used to understand the situations when the modifications required in the software system affect large segments of software code or several components of the software.
2. It helps identifying the relationship among the components that are affected with the change and thereby helping to understand the overall software structure.
3. It is used to record the history of modification, which helps in maintaining quality in the software system.

Software Rejuvenation

Sometimes, organizations have to take difficult decisions about how to make their systems more maintainable. The choices may include enhancing or completely replacing a software system. Note that each choice has the same objective, that is, to preserve or increase the software quality while keeping the costs low. Software rejuvenation is a maintenance technique which helps in taking appropriate decisions.

Software rejuvenation checks the system's work products in order to extract additional information or to reformat them in order to make these work products more understandable. Generally, four types of software rejuvenation exist, namely, re-documentation, restructuring, reverse engineering, and reengineering. Re-documentation uses static analysis of the source code to produce additional information, which helps the software maintenance team to understand and refer to the code. In source code, component size, component calls, calling parameters, and control

paths are examined to understand what and how code does it. The output of static code analysis is either graphical or textual, which can be used to assess whether the re-documentation is required.

Restructuring

Restructuring involves the transformation of unstructured code into structured code thereby making it easier to understand and change. Restructuring involves the following steps.

1. Static analysis is performed, which provides information that is used to represent code as a directed graph or associative (semantic) network. The representation mayor may not be in a human readable form; thus, an automated tool is used.
2. Transformational techniques are used to refine (simplify) the representation.
3. Refined representation is interpreted and used to generate the structured code.

Reverse Engineering

Reverse engineering like re-documentation, focuses on providing information about the specification and design information using the software code. The information extracted from specification and design is stored in a format that can be easily modified. Reverse engineering is a useful technique when the software maintenance team is unable to understand the processes involved in the software system. Reverse engineering involves the following steps.

1. Source code is collected with the help of an automated tool used for reverse engineering. This tool is used to represent the structure and the naming information of variables, functions and other components in the software code.
2. Static analysis is performed.
3. Some methods such as standards structured analysis and design methods are used. These methods are used to extract information such as data dictionaries, data-flow, control flow, and entity relationship (ER) diagrams for the reverse engineering technique.

The advantages associated with reverse engineering are listed below.

1. It focuses on recovering the lost information from the programs.
2. It provides the abstract information from the detailed source code implementation.
3. It improves system documentation 'that is either incomplete or out of date.
4. It manages the complexity that is present in the software programs.
5. It detects the adverse effects of modification in the software system.

Re-engineering

Re-engineering is an extension of reverse engineering. This technique refers to the systematic transformation of the present software system into a new form to make quality improvements in operation, system capability, functionality, and achieving high performance at low costs.

Re-engineering involves the following steps.

1. The system is reverse engineered and represented internally for human and computer modifications.
2. The software system is corrected and completed. This includes updating internal specification and design.
3. Using new specification and design, a new system is generated.

Advantages

Reduced cost: Generally, it is observed that the software systems that are maintained using re-engineering incur less cost as compared to developing the software system all over again.

Reduced risk: The incremental nature of re-engineering means that the existing staff skills evolve as the software system evolves. Due to this fact, the risks associated with the modifications in the software system are reduced.

Better use of existing staff: The individuals who worked on software maintenance can be retained while the re-engineering technique is being used. In addition, the staff can be extended to accommodate new skills during reengineering. Due to this fact, the re-engineering technique has less number of risks and incurs less expenditure while hiring the new staff.

Incremental development: Re-engineering techniques can be carried out in stages according to the availability of budget and resources. This technique is useful in operational organizations with working software systems. In such organizations, the staff can easily adapt to the re-engineered software system.

Data corruption

Data corruption refers to errors in computerdata that occur during writing, reading, storage, transmission, or processing, which introduce unintended changes to the original data. Computer, transmission, and storage systems use a number of measures to provide end-to-end data integrity, or lack of errors.

In general, when data corruption occurs a file containing that data will produce unexpected results when accessed by the system or the related application. Results could range from a minor loss of data to a system crash. For example, if a document file is corrupted, when a person tries to open that file with a document editor they may get an error message, thus the file might not be opened or might open with some of the data corrupted (or in some cases, completely corrupted, leaving the document unintelligible). The adjacent image is a corrupted image file in which most of the information has been lost.

Some types of malware may intentionally corrupt files as part of their payloads, usually by overwriting them with inoperative or garbage code, while a non-malicious virus may also unintentionally corrupt files when it accesses them. If a virus or trojan with this payload method manages to alter files critical to the running of the computer's operating system software or physical hardware, the entire system may be rendered unusable.

Some programs can give a suggestion to repair the file automatically (after the error), and some programs cannot repair it. It depends on the level of corruption, and the built-in functionality of the application to handle the error.

Causes of data corruption and loss

Common causes of data corruption and loss include:

- Power outages or other power-related problems.
- Improper shutdowns, such as caused by power outages or performing a hard restart: pressing and holding the power button or, on Macs so equipped, the restart button.
- Hardware problems or failures, including hard drive failures, bad sectors, bad RAM, and the like.
- Failure to eject external hard drives and related storage devices before disconnecting them or powering them off.
- Bad programming, particularly if it results in either hard restarts or data that is saved incorrectly.

Any of these causes can result in a corrupted hard drive directory. A corrupted hard drive directory can cause files to apparently "go missing" and lead to further data loss or corruption, such files being overwritten with new data as a corrupted directory may no longer accurately reflect what disk space is free or available vs. the disk space that contains data. The term *data* is used here to mean both files you have created as well as application and operating system code. Technologies such as File System Journaling have helped to reduce the potential for directory corruption due to power outages or hard restarts, but journaling is not foolproof. Likewise, while hard drives have become exceedingly reliable, they are still known to fail catastrophically with little or no warning. If operating system files become corrupted, your Mac may not start up or experience recurring kernel panics when a corrupted kernel extension is used. In addition to the causes cited above, operating system files can be corrupted by failed Software Updates. Accordingly, it is appropriate to implement certain strategies to minimize the risks of data corruption and loss.

	UNIT 1	

S.NO	QUESTION	OPTION A	OPTION B	OPTION C	OPTION D	ANSWER
1	_____ is a logical rather than a physical element.	software	hardware	network	Mother board	software
2	The foundation for engineering is _____ __	methods	tools	process	software	process
3	Identify the water fall model from the options given below.	Linear	Sequential	Prototyping	RAD	Linear
4	Identify the model that emphasizes on a short development cycle	Linear	Sequential	Prototyping	RAD	RAD
5	_____ software is written to service other programs	Real time	System	Business	Embedded	System
6	_____ is an evolutionary process model.	Incremental	Spiral	Linear	Waterfall	Spiral
7	The use of 4GT without design will cause _____	poor customer acceptance	maintainability	unacceptable quality	difficult to customize	poor customer acceptance
8	Engineering based activity begins with _____	Identification of candidate class	Examining the data	application to accompolish the manipulation	class package	Identification of candidate class

9	Identify the direct measure from the options	Functionality	quality	efficiency	execution speed	execution speed
10	It is derived by normalizing quality and / or productivity measures by considering the size of the software that is to be produced	Function oriented	Size oriented	Object Oriented	LOC	Size oriented
11	Identify the option that is not used to achieve reliability of cost and effort	Delay estimation till project is complete	Estimate on related projects	Use techniques without decomposing the projects	Use empirical models	Use techniques without decomposing the projects
12	FP in software quality measurement means_____	Frequent patterns	Function point	First process	Function quality	Function point
13	In this approach, the planner must identify the type of application, establish its magnitude on a qualitative scale, and then refine the	Function point	Fuzzy logic	Change sizing	Component sizing	Fuzzy logic

	magnitude within the original range.					
14	Change sizing encompasses the use of existing software	Yes	No			Yes
15	Identify the ways that LOC and FP use the data for software project estimation.	As an estimation variable to assess cost	as baseline metric from current projects	As a baseline metric from past projects	as an estimation variable to modify	As a baseline metric from past projects
16	If historical data indicates that 10 lines per report is required, what would be planners estimate of LOC that will be required for 100 such reports	10	100	110	1000	1000
17	It refers to the degree of performance of the software against its intended purpose.	Functionality	Efficiency	Reliability	Maintainability	Functionality
18	It refers to the ability of the software to use system resources in the most effective and efficient manner.	Functionality	Efficiency	Reliability	Maintainability	Efficiency

19	_____ is the capability of a software to maintain its level of performance under the given condition for a stated period of time.	Functionality	Efficiency	Reliability	Maintainability	Reliability
20	_____ means, the ease with which the modifications can be made in a software system to extend its functionality, improve its performance, or correct errors	Functionality	Efficiency	Reliability	Maintainability	Maintainability
21	_____ means the ability of software to be transferred from one environment to another, without or minimum changes.	Functionality	Efficiency	Reliability	Portability	Portability
22	Process based estimation begins with _____	Delineation of software functions	Effort	average labour rates	average labour rates	Delineation of software functions

23	Identify the approach to software sizing that use the reasoning technique	Function point	Standard component	Fuzzy logic	Change sizing	Fuzzy logic
24	Automated tools implement decomposition technique or empirical models	Yes	No			Yes
25	PSP follows _____ approach.	Evolutionary	systematic	disciplined	quantifiable	Evolutionary
26	People with software responsibility are often under pressure to maintain budgets, keep schedules from slipping, and improve quality. Identify the myth here.	Customer	Management	Practitioner	Mongolian horde	Management
27	The foundation for software engineering is the _____.	tools	Methods	Quality focus	process layer	process layer
28	RAD stands for	Relative Application Development	Rapid Application Development	Rapid Application Document	None of the mentioned	Relative Application Development
29	SDLC stands for _____	Software Development Life Cycle	System Development Life cycle	Software Design Life Cycle	System Design Life Cycle	Software Development Life Cycle

30	Which model can be selected if user is involved in all the phases of SDLC?	Waterfall Model	Prototyping Model	RAD Model	Prototyping Model & RAD Model	RAD Model
31	The "Bed Rock" that supports software Engineering is_____.	tools	Methods	Quality focus	process layer	Quality focus
32	Identify the software process model that involves a systematic progression through analysis, design, coding, testing and maintenance phases.	Waterfall model	Prototyping Model	RAD model	Incremental process	Waterfall model
33	Identify the model that emphasizes an extremely short development cycle.	Waterfall Model	Prototyping Model	RAD Model	Incremental process	RAD Model
34	Identify the model that combines elements of linear and parallel process flows.	Waterfall model	Prototyping Model	RAD model	Incremental process	Increment al process

35	Identify the software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the waterfall model.	Spiral model	Prototyping Model	RAD model	Incremental process	Spiral model
36	It provides the potential for rapid development of increasingly more complete versions of the software.	Spiral model	Prototyping Model	RAD model	Incremental process	Spiral model
37	It is evolutionary in nature, demanding an iterative approach to the creation of software.	Component-based model	4GT model	CMM	PCMM	Component-based model
38	Identify the model that is based on Non-Procedural Language techniques.	Component-based model	4GT model	CMM	PCMM	4GT model

39	Organizations at this level are characterized by a tendency to over commit, abandon processes in the time of crisis, and not be able to repeat their past successes.	Maturity level 1	Maturity Level 2	Maturity Level 3	Maturity Level 4	Maturity level 1
40	Identify the level where the requirements, processes, work products, and services are managed.	Maturity level 1	Maturity Level 2	Maturity Level 3	Maturity Level 4	Maturity Level 2
41	Identify the level when processes are well characterized and understood, and are described in standards, procedures, tools, and methods.	Maturity level 1	Maturity Level 2	Maturity Level 3	Maturity Level 4	Maturity Level 3

42	At this level processes are managed more proactively using an understanding of the interrelationships of the process activities and detailed measures of the process, its work products, and its services.	Maturity level 1	Maturity Level 2	Maturity Level 3	Maturity Level 4	Maturity Level 3
43	At this level sub processes are selected that significantly contribute to overall process performance.	Maturity level 1	Maturity Level 2	Maturity Level 3	Maturity Level 4	Maturity Level 4
44	This maturity level focuses on continually improving process performance through both incremental and	Maturity Level 2	Maturity Level 3	Maturity Level 4	Maturity Level 5	Maturity Level 5

	innovative technological improvements.					
45	It is a process level improvement training and appraisal program.	ISO	CMMI	CMM	PCMM	CMMI
46	Recognize the maturity framework model that focuses on continuously improving the management and development of the human assets of a software.	PSP	TSP	CMM	PCMM	PCMM
47	This process emphasizes personal measurement of both the work product that is produced and the resultant quality of the work.	PSP	TSP	CMM	PCMM	PSP
48	The goal of this software process is to build a “self	PSP	TSP	CMM	PCMM	TSP

	directed" project team that organizes itself to produce high-quality software.					
49	Recognize the framework activity where code is generated, reviewed, compiled, and tested.	Post mortem	Development	Planning	High level design	Development
50	Identify the framework activity where defect estimate is made.	Post mortem	Development	Planning	High level design	Planning
51	Recognize the process in which a new product is developed by reverse engineering an existing product.	Six Zigma	SPICE	Clean room	SCAMPI	Clean room
52	This level determines the effectiveness of the process using the measures and metrics collected.	Post mortem	Development	Planning	High level design	Development

53	The primary objective of this model is to improve the capability of the workforce.	PSP	TSP	CMM	PCMM	PCMM
54	In this model developers often make implementation compromises.	Component-based	Prototyping	CMM	PCMM	Prototyping
55	It is a rigorous and disciplined methodology that uses data and statistical analysis to measure and improve a company's operational performance by identifying and eliminating defects.	SEI CMMI	CBA IPI	SPICE	SCAMPI	SCAMPI
56	It is the capability of software to maintain its level of performance under the given condition for a stated period of time.	Reliability	Accuracy	Compliance	interoperability	Reliability

57	It is a well-defined evolutionary plateau toward achieving a mature software process.	Maturity level	Quality level	ISO level	Team software level	Maturity level
58	Identify the framework activity that is not part of PSP model.	Post mortem	Development	Planning	Reuse	Reuse
59	Identify the step that is not a core step in Six Sigma methodology.	Define	Measure	Analyze	Assess	Assess
60	Which is the first step in the software development life cycle?	Analysis	Design	Problem/Opportunity Identification	Development and Documentation	Problem/Opportunity Identification

	UNIT 2					
	18CCP301 SOFTWARE MODELS AND ENGINEERING					
	QUESTION	OPTION A	OPTION B	OPTION C	OPTION D	ANSW
1	The most desirable form of cohesion is	logical cohesion	procedural cohesion	functional cohesion	temporal cohesion	procedu
2	The most desirable form of coupling is	(a)control coupling	(b) data coupling	(c)common coupling	(d) content coupling	(b) data
3	COCOMO was developed initially by	(a) B.W.Bohem	(b) Gregg Rothermal	(c) B.Beizer	(d) Rajiv Gupta	(a) B.W
4	Identify the elements of software project management from the options	People, Product, Process, Project	Principles, People, Product, Price	Price, People, Process, Perks	People, Product, Process, Plan	People, Project
5	Recognize the direct measure of the software process among the following options	Functionality	Percentage of Quality	Reliabilty	Execution Speed	Execut
6	Recognize the indirect measure of the software process among the following options	Functionality	Memory size	Defects reported	LOC	Functio
7	Discover the metric widely used for functionality oriented metric.	Function oriented	Number of Scenario scripts	c) Key classes	d) LOC	Functio
8	It is a form of problem solving where the problem to be solved is too complex to be	Decomposition	Comprehensiveness	Thoroughness	Completeness	Decom

	considered in one piece					
9	Expand COCOMO.	Constructive Cost model	Conversion cost model	Complete cost model	Comprehensive cost model	Constru
10	This system model depicts the whole function as a single transformation.	context model	behaviour model	data model	waterfall model	context
11	Discover the model that indicates how software will respond to external events or stimuli.	context model	behaviour model	data model	waterfall model	behavio
12	_____ is a component of behaviour model.	DFD	UML	ERD	USE CASE	UML
13	_____ represents active states for each class and the events (triggers) that cause changes between these active states.	DFD	UML	ERD	USE CASE	UML
14	_____ represents all data objects that are entered, stored, transformed, and produced within an application.	DFD	UML	ERD	USE CASE	ERD
15	It is a representation of composite information that must be understood by software.	Data attributes	Relationships	Data objects	Design process	Data ob

16	_____ is an iterative process through which requirements are translated into a blueprint" for constructing the software.	Software design	Relationships	Data objects	Design process	Softwa
17	_____is an indication of the relative functional strength of a module.	Cohesion	Relationships	Data objects	Coupling	Cohesi
18	*Coupling* is an indication of the relative interdependence among modules.	Cohesion	Relationships	Data objects	Coupling	Couplin
19	A _____should (ideally) do just one thing.	cohesive module	Relationships	Data objects	Coupling module	cohesiv
20	_____ focuses primarily on classes that are extracted directly from the statement of the problem.	Requirements modeling	Design modeling	Testing modeling	code modeling	Require
21	_____ greatly reduces the effort required to extend the design of an existing object-oriented system.	Inheritance	Polymorphism	Data abstraction	none of these	Polymo
22	Characteristics of an existing class can be overridden, and different versions of attributes or operations are implemented for the new class. What is implemented	Inheritance	Polymorphism	Data abstraction	none of these	Inherita

	here?					
23	A _____ provides a static or structural view of a system.	class diagram	use case diagram	ERD	DFD	class di
24	_____ are the icons used to represent classes and interfaces in UML.	Boxes	Oval	Pointers	dashed lines	Boxes
25	In UML, if the arrow points from to a class with a solid line. What does it denote?	A class is a subclass of another class	implementation of an interface	Public visibility	Private visibility	A class another
26	What does a leading minus sign (–) in UML denote?	A class is a subclass of another class	implementation of an interface	Public visibility	Private visibility	Private
27	A *dependency* relationship represents another connection between classes and is indicated by a _____.	Solid line	Oval	Pointers	Dashed lines	Dashed
28	An _____is a special kind of association indicated by a hollow diamond on one end of the icon.	*aggregation*	*composition*	*generalization*	*authorization*	*aggreg*
29	Every software development project contains elements of uncertainty. This is known as _____	project risk	module risk	temporary risk	software risk	project

30	Risk management means risk _____.	both containment and mitigation	mitigation action	containment action	Risk analysis	both co mitigat
31	The use of _____ technologies often leads directly to project failure.	new or unproven technologies	complex user and functional requirements	Non performing hardware	proven technologies	new or technol
32	It is measured by benchmarks and threshold testing throughout the project to ensure that the work products are moving in the right direction.	performance	organizational objective	Architecture requirements	efficiency	perform
33	_____ should find a balance between the needs of the development team and the expectations of the customers.	performance	organizational	Architecture	Module level	organiz
34	The _____ outlines the response that will be taken for each risk.	monitoring plan	risk management plan	mitigating plan	Risk mitigation plan	risk ma
35	Revise risk plans according to any major changes in project schedule is done during _____.	mitigation	monitoring	containment	Risk analysis	mitigat
36	When a risk occurs, the corresponding mitigation response should be taken from the _____	monitoring plan	risk management plan	mitigating plan	Risk mitigation plan	risk ma

37	Identify the mitigating action when a risk is impacting the project	accept	monitor	contain	avoid	accept
38	Taking an action to minimize the impact or reduce the intensification of the risk is _____.	control	transfer	contain	avoid	control
39	Implementing an organizational shift in accountability, responsibility, or authority to other stakeholders that will accept the risk is _____	control	transfer	continue monitoring	avoid	transfe
40	Adjusting the project scope, schedule or constraints to minimize the effects of the risk is_____ .	control	transfer	continue monitoring	avoid	avoid
41	Sharing information and getting feedback about risks is _____.	control	transfer	continue monitoring	communicate	commu
42	Which metrics are derived by normalizing quality and/or productivity measures by considering the size of the software that has been produced?	size oriented	function oriented	object oriented	use case oriented	size ori

43	What is the most common measure for correctness?	defects per kloc	errors per kloc	$ per kloc	page documentation per kloc	defects
44	Line of code(LOC) of the product comes under which type of measures?	Direct measures	Indirect measures	Coding	Testing	Direct m
45	Which among these best represents Coupling for an ideal device?	Do exactly one job completely	Be loosely coupled to the rest of the program	Hide its Implementation	Never change its interface	Be loos rest of t
46	47. _____ is a measure of the degree of interdependence between modules.	Cohesion	Coupling	Interconnection	Interrelation	Couplin
47	Which risk gives the degree of uncertainty and the project schedule will be maintained so that the product will be delivered in time?	Business risk	Technical risk	Schedule Risk	Project risk	Schedu
48	LOC based estimation techniques require problem decomposition based on	Information domain values	Project schedule	Software functions	Process activities	Informa
49	Process based estimation techniques require problem decomposition based on	Information domain values and project schedule	Process schedule only	Software functions and Process activities	Project Schedule	Softwar Process
50	Coupling is a qualitative indication of the	can be written more compactly	focuses on just one thing	is able to complete a function in a timely manner	is connected to other modules and the outside	focuses

	degree to which a module				world	
51	Cohesion is a qualitative indication of the degree to which a module	can be written more compactly	focuses on just one thing	is able to complete a function in a timely manner	is connected to other modules and the outside world	is conn module world
52	What encapsulates both data and data manipulation functions ?	Object	Class	Super Class	Sub Class	Object
53	Which of the following is a mechanism that allows several objects in a class hierarchy to have different methods with the same name?	Aggregation	Polymorphism	Inheritance	Abstraction	Polymo
54	Identify the sizing approach that uses reasoning techniques.	Fuzzy logic	Change sizing	Function point sizing	Standard component sizing	Fuzzy lo
55	Identify the approach when a project encompasses the use of existing software that must be modified in some way as part of a project	Fuzzy logic	Change sizing	Function point sizing	Standard component sizing	Functio
56	Identify the system model that depicts the security function as a single transformation	Context	Process	Product	DFD Level 1	Contex
57	What assess the risk and your plans for risk	Risk monitoring	Risk planning	Risk analysis	Risk identification	Risk mo

	mitigation and revise these when you learn more about risk?					
58	The worst type of coupling is	Data coupling	Control coupling	Stamp coupling	Content coupling	Conten
59	_____ is a central modeling technique that runs through nearly all object-oriented methods.	class diagram	component diagram	object diagram	package diagram	class di
60	_____ is also know as Successive version model.	ncremental process model	Component based model	Spiral model	4GT	ncreme

	UNIT 3			
			18CCP301 SOFTWARE MODELS AND ENGINEERING	
	QUESTION	OPTION A	OPTION B	OPTION C
1	One weakness of boundary value analysis and equivalence partitioning is	They are not effective	They do not explore combinations of input circumstances	They explore combinations of input circumstances
2	During the development phase, the following testing approach is not adopted	Unit testing	Bottom up testing	Integration testing
3	KPA in CMM stands for	Key Process Area	Key Product Area	Key Principal Area
4	Which one of the following is not a functional requirement ?	a) Maintainability	b) Portability	c) Robustness
5	"Consider a system where, a heat sensor detects an intrusion and alerts the security company." What kind of a requirement the system is providing ?	Functional	Non-Functional	Known Requirement
6	This requirement document does not include details of the system architecture or design.	user	system	functional
7	It is a complete and detailed specification of the	user	system	functional

	whole system.			
8	Recognize the document that captures complete description about how the system is expected to perform	SRS	SRA	UCD
9	Software requirement can be _____	system	Non-Functional	Known Requirement
10	The intent of project metrics is	minimization of development schedule	for strategic purposes	assessing project quality on ongoing basis
11	The user system requirements are the parts of which document?	SDD	SRS	DDD
12	Which is one of the most important stakeholder from the following ?	Entry level personnel	Managers	Middle level stakeholder
13	Which of the following is not defined in a good Software Requirement Specification (SRS) document?	Functional requirement	Nonfunctional requirement	Goals of implementation
14	Software Requirement Specification (SRS) is also known as specification of _____.	White box testing	Black box testing	Integrated testing
15	Which document is created by system analyst after the requirements are	SRS	user documentation	Software design documentation

	collected from Various stakeholders			
16	Which of the following is not a direct measure of SE process?	Efficiency	Cost	Effort Applied
17	The amount of time that the software is available for use is known as	Reliability	Usability	Efficiency
18	Which of the following is a direct measure of product?	LOC	Complexity	Reliability
19	In size oriented metrics, metrics are developed based on the _____	number of Functions	number of user inputs	number of lines of code
20	User Acceptance testing is	White box testing	Black box testing	Gray box testing
21	Error guessing is a	Test verification techniques	Test execution techniques	Test control management techniques
22	Which of the following is not a part of test plan?	Scope	Mission	Objective
23	A set of activities that ensure that software correctly implements a specific function.	verification	testing	implementation
24	What do you call testing individual components?	system testing	unit testing	validation testing
25	A testing strategy that test the application as a whole.	Requirement Gathering	Verification testing	Validation testing
26	Which of the following is NOT a white box technique?	Statement testing	Path testing	State transition testing
27	Which of the following would NOT normally form part of a test plan?	Features to be tested	Risks	Incident reports

28	What is the main difference between a walkthrough and an inspection?	A walkthrough is lead by the author, whilst an inspection is lead by a trained moderator.	An inspection has a trained leader, whilst a walkthrough has no leader.	Authors are not present during inspections, whilst they are during walkthroughs.
29	Which one of the following statements about system testing is NOT true?	System tests are often performed by independent teams.	Functional testing is used more than structural testing.	Faults found during system tests can be very expensive to fix.
30	What is testing process' first goal?	Bug prevention	Testing	Execution
31	Software mistakes during coding are known as	errors	failures	bugs
32	Which of the following is not a part of bug report?	Test case	Output	Software Version
33	Which is a black box testing technique appropriate to all levels of testing?	Acceptance testing	Regression testing	Equivalence partitioning
34	Effective testing will reduce _____ cost.	maintenance	design	coding
35	Size and Complexity are a part of	Product Metrics	Process Metrics	Project Metrics
36	Cost and schedule are a part of	Product Metrics	Process Metrics	Project Metrics
37	Boundary value analysis belong to?	White Box Testing	Black Box Testing	White Box & Black Box Testing
38	Quality Management in software engineering is also known as	SQA	SQM	SQI
39	What is Six Sigma?	It is the most widely used strategy for statistical quality assurance	The "Six Sigma" refers to six standard deviations	It is the most widely used strategy for statistical quality assurance AND The "Six Sigma" refers to six standard deviations

40	Non-conformance to software requirements is known as	Software availability	Software reliability	Software failure
41	Why is software difficult to build ?	Controlled changes	Lack of reusability	Lack of monitoring
42	What is validating the completeness of a product?	Identification	Software	Auditing and Reviewing
43	_____ methods can be used to drive validations tests	Yellow-box testing	Black-box testing	White-box testing
44	Which of the following is black-box oriented and can be accomplished by applying the same black-box methods discussed for conventional software?	Conventional testing	OO system validation testing	Test case design
45	In which of the following testing strategies, a smallest testable unit is the encapsulated class or object?	Unit testing	Integration testing	System testing
46	Which of the following testing types is not a part of system testing?	Recovery testing	Stress testing	System testing
47	Which is not a desirable characteristic of SRS?	Concise	Ambiguous	Traceable
48	Which of the following is not included in SRS document?	Functional requirements	Non Functionl Requirements	Goals of implementation
49	Which of the quality of SRS document compares th eresuolts of the phase with another phase?	Structured	Verifiable	Traceable
50	If every requirement stated in the Software Requirement	Unambiguous	Consistent	Verifiable

	Specification (SRS) has only one interpretation, SRS is said to be correct _____ .			
51	Quality of the product comes under which type of measures?	Indirect measures	Direct measures	Coding
52	During software development which factor is most crucial ?	People	Process	Product
53	Milestones are used to ?	Know the cost of the project	Know the status of the project	Know the user expectations
54	White box testing, a software testing technique is sometimes called ?	Basic path	Graph Testing	Dataflow
55	Black box testing sometimes called ?	Data Flow testing	Loop Testing	Behavioral Testing
56	The objective of testing is ?	Debugging	To uncover errors	To gain modularity
57	Context diagram explains	The overview of the system	The internal view of the system	The entities of the system
58	1. Which is not a characteristic of a good SRS?	Correct	Complete	Brief
59	1. Outcome of requirements specification phase is	Design Document	Test Document	UML Diagram
60	Which one is a functional requirement?	Scalability	Reliability	Authenticate

	UNIT 4			
	18CCP301 SOFTWARE MODELS AND EN			
	QUESTION	OPTION A	OPTION B	OPTION C
1	It is referred to any program or utility that performs a monitoring function.	Snooping	Recoverability	Prototypi
2	Identify the test plan that typically addresses multiple test levels	Test plan	Master test plan	Phase test
3	Identify the testing plan done at a specific level of testing from the options below	unit test plan	security test plan	Integration
4	_____is the forced failure of the software to verify if the recovery is successful.	Recovery testing	failure testing	verification
5	Which of the following is not project management goal?	Keeping overall costs within budget	Delivering the software to the customer at the agreed time	Maintaining and well-fun developmen
6	Identify the software testing type that is performed to determine whether operations can be continued after a disaster or after the integrity of the system has been lost.	recovery testing	validation testing	Black box tes
7	_____ is functional testing.	Recovery testing	System testing	stability test
8	The open source movement has meant that there is a huge	free of cost	low cost	high cost

	reusable code base available at			
9	COTS stands for	Commercial Off-The-Shelf systems	Commercial Off-The-Shelf states	Commercial System state
10	Which of the following is not an advantages of software reuse?	lower costs	faster software development	high effectiv
11	A special case of software reuse is ……. where a whole application system is reused by implementing it across a range of different computers and operating systems.	Application System Reuse	Generator Based Reuse	Domain Orie
12	……. involves analyzing the system state to gauge the extent of the state corruption.	Exception Handling	Defensive Programming	Failure Prev
13	……. is the process of modifying the state space of the system so that the effects of the fault are minimized.	Fault avoidance	Fault detection	Fault repair
14	The modification of the software to match changes in the ever changing environment, falls under which category of software maintenance?	Corrective	Adaptive	Perfective
15	What type of software testing is generally used in Software Maintenance?	Regression Testing	System Testing	Integration T
16	Which selective retest technique selects every test case that causes a modified program to produce a different output than its original version?	Coverage	Minimization	Safe

17	_____ measures the ability of a regression test selection technique to handle realistic applications.	Efficiency	Precision	Generality
18	Which regression test selection technique exposes faults caused by modifications?	Efficiency	Precision	Generality
19	Which one of the following is not a maintenance model?	Waterfall model	Reuse-oriented model	Iterative enh model
20	Which of the following manuals is not a user documentation?	Beginner's Guide	Installation guide	Reference G
21	How many stages are there in Iterative-enhancement model used during software maintenance?	two	three	four
22	Which subsystem implements the requirements defined by the application?	UI	DBMS	Application s
23	"A client is assigned all user presentation tasks and the processes associated with data entry".Which option supports the client's situation?	Distributed logic	Distributed presentation	Remote pres
24	What is used to pass SQL requests and associated data from one component to another?	Client/server SQL interaction	Remote procedure calls	SQL Injection
25	Which of the following services is not provided by an object?	Activating & Deactivating Objects	Security features	Files implem entities iden within the EI
26	Which of the following term is best defined by the statement:"When one object invokes another independent object, a message is passed between the	Control couple	Application object	Data couple

	two objects."?			
27	CORBA stands for_____.	Common Object Request Build Architecture	Common Object Request Broker Architecture	Common Ob Request Brea Architecture
28	RMI stands for?	Remote Mail Invocation	Remote Message Invocation	Remaining M Invocation
29	A typical _____ program creates some remote objects, makes references to these objects accessible, and waits for clients to invoke methods on these objects.	Server	Client	Thread
30	A typical _____ program obtains a remote reference to one or more remote objects on a server and then invokes methods on them.	Server	Client	Thread
31	The _____ layer, which provides the interface that client and server application objects use to interact with each other.	Increasing	Count	Bit
32	A layer which is the binary data protocol layer.	stub layer	skeleton layer	remote laye
33	A middleware layer between the stub skeleton and transport.	remote layer	instruction layer	reference la
34	An object acting as a gateway for the client side.	skeleton	stub	remote
35	A gateway for the server side object.	skeleton	stub	remote
36	RMI uses stub and skeleton for communication with the _____ object.	client	remote	server

37	File transfer protocol (FTP) is built on _____architecture	peer to peer	client server	CLIENT TO C
38	The time taken by a packet to travel from client to server and then back to the client is called ____	STT	RTT	PTT
39	The first line of HTTP request message is called ____.	request line	header line	status line
40	Cyclomatic Complexity cannot be applied in _____.	Re-engineering	Risk Management	Test Planni
41	Verification is the responsibility of _____.	Developer	Designer	Tester
42	Which activity is carried out first?	Verification	Validation	Maintenanc
43	Which of the following is / are not a verification activity?	Inspection	Testing	Walkthroug
44	Which is the odd one out?	Error Guessing	Walkthrough	Data flow a
45	Which of the nfollowing are advantages of using LOC as a size orientes nmetric?	LOC is easily computed	LOC is a language dependent measure	LOC is a la independen
46	Validation is focused on _____.	Product	Process	Risk
47	What are prototypes ?	Prototypes is a working model of part or all of a final product	Prototypes does not represent any sort of models	Prototype c consist of ful
48	What are the notations for the Use case Diagrams ?	Use case	Actor	Prototype
49	White box testing can be started	After SRS creation	After designing	After progr
50	The type of software testing in which each model is tested along	Integration	Acceptance	Mutation

	in an attempt to discover any errors in code			
51	Which of the following is a possible benefit of independent testing?	Independent testers see other and different defects and are unbiased.	Independent testers do not need extra education and training.	Independent reduce the b in the incide managemen
52	…… are used for risk-based testing where testing is directed to areas of greatest risk.	Analytical approaches	Model-based approaches	Methodical
53	……. describes any guiding or corrective actions taken as a result of information and metrics gathered and reported.	Test control	Test monitoring	Test reporti
54	……. is concerned with summarizing information about the testing endeavor.	Test control	Test monitoring	Test reportin
55	The purpose of …… is to provide feedback and visibility about test activities.	Test control	Test monitoring	Test report
56	The purpose of …… is to establish and maintain the integrity of the products of the software or system through the project and product lifecycle.	Test control	Test monitoring	Test reporti
57	Which of the following risk does NOT include product risks in software testing?	Failure-prone software delivered	Software that does not perform its intended functions	Low quality design and c
58	For testing …….. may involve ensuring all items of test ware are identified, version controlled, tracked for changes, related to each other and related to the development	Test control	Test monitoring	Test reportin

	items, so that traceability can be maintained throughout the test process.			
59	The testing in which code is checked	Black box testing	White box testing	Red box tes
60	Unit testing is done by	Users	Developers	Customers

	UNIT 5		
			18CCP301 SOFTWARE MO
	QUESTION	OPTION A	OPTION B
1	Reverse engineering of data focuses on from the options below	Internal data structures	Quality
2	The process of generating analysis and design documents is known as	Software engineering	Software re-engineering
3	The process of transforming a model into source code is known as	Forward engineering	Reverse engineering
4	Which of these benefits can be achieved when software is restructured?	Higher quality programs	Reduced maintenance effort
5	What are the problems with re-structuring?	Loss of comments	Loss of documentation
6	Which of the following is not an example of a business process?	designing a new product	hiring an employee
7	In reverse engineering process, what refers to the sophistication of the design information that can be extracted from the source code?	interactivity	completeness
8	In reverse engineering, what refers to the level of detail that is provided at an abstraction level?	interactivity	completeness
9	The core of reverse engineering is an activity called	restructure code	directionality
10	Forward engineering is also known as	extract abstractions	renovation

11	Reverse engineering is the process of deriving the system design and specification from its	GUI	Database
12	Architecture description language represent architectural structures that can be divided into which of the following?	Static	Dynamic
13	Why is Requirements Management Important ? It is due to the changes	to the environment	in technology
14	Which technique is applied to ensure the continued evolution of legacy systems ?	Forward engineering	Reverse Engineering
15	Which of the following items are not measured by software project metrics?	Inputs	Outputs
16	Which of the following is not a module type?	Object modules	Hardware modules
17	Which of the following is a data problem?	hardware problem	record organisation problems
18	When does one decides to re-engineer a product?	when tools to support restructuring are disabled	when system crashes frequently
19	19. In reverse engineering process, what refers to the sophistication of the design information that can be extracted from the source code?	interactivity	completeness
20	Which of the following is the task of project indicators:	help in assessment of status of ongoing project	track potential risk
21	Consider the example and categorize it accordingly, “A pattern-matching system developed as part of a text-processing system may be reused in a database management system”.	Application system reuse	Component reuse
22	Which of the following is a generic structure that is extended to create a more specific subsystem or application?	Software reuse	Object-oriented programming language

23	Which of the following is not an advantages of software reuse?	lower costs	faster software development
24	Which of the following is not a metric for design model?	Interface design metrics	Component-level metrics
25	Architectural Design Metrics are _____ in nature.	Black Box	White Box
26	The process each manager follows during the life of a project is known as	Project Management	Manager life cycle
27	Milestones are used to ?	Know the cost of the project	Know the status of the project
28	a new _____ is defined when major changes have been made to one or more configuration objects	entity	item
29	which of the following activity is not part of a software reengineering process model?	forward engineering	inventory analysis
30	which of the model is not to be considered when reverse engineering	abstraction level	completeness
31	the first reverse enginnering activity involves seeking to understand	data	processing
32	When does one decides to re-engineer a product?	when tools to support restructuring are disabled	when system crashes frequently
33	In reverse engineering process, what refers to the sophistication of the design information that can be extracted from the source code?	interactivity	completeness
34	In reverse engineering, what refers to the level of detail that is provided at an abstraction level?	interactivity	completeness

35	Which of the following is not included in Architectural design decisions?	a) type of application	b) distribution of the system
36	Which one of the following models is not suitable for accommodating any change?	Build & Fix Model	Prototyping Model
37	it is a tangible by-product produced during the development of software.	artifacts	prototype
38	_____ is a form of perfective maintenance that modifies the structure of a program's source code.	Software restructuring	Reverse Engineering
39	Xthis model does not attempt to start with a full specification of requirements. Instead, development begins by specifying and implementing just part of the software	iterative life cycle model	water fall model
40	It refers to the process of reinventing the business processes	Software restructuring	Reengineering
41	Is a software design approach for the development of software systems	Model-driven architecture	Reengineering
42	_____is a set of required activities and the outcome of the activities with a target to produce a software product.	Software Process	Software restructuring
43	Failure to meet _____ indicates that a project is not proceeding to plan and usually triggers corrective action by management.	Milestone	Error detection
44	_____ is a software development methodology, which favors iterative development and the rapid construction of prototypes instead of large amounts of up-front planning.	RAD	Spiral
45	The_____ is a sequential development approach, in which development is seen as flowing steadily downwards	waterfall model	Spiral Model

46	_____ is a structure imposed on the development of a software product.	Software Process	Software restructuring
47	The _____ provides a mechanism for managing everyone's expectations throughout the project lifecycle.	periodic Status Assessment	SDLC
48	An iterative process of software development where requirements are continually revised	waterfall model	iterative model
49	These systemwide events are held at the end of each development phase.	Major milestones	Minor mile stones
50	These iteration-focused events are conducted to review the content of an iteration in detail and to authorize continued work.	Major milestones	Minor mile stones
51	These periodic events provide management with frequent and regular insight into the progress being made.	Major milestones	Minor mile stones
52	Quality Indicators indicate _____ .	work and progress	breakage and modularity
53	Management Indicators indicate _____ .	work and progress	budgeted cost
54	_____ during the engineering stage is geared mostly toward establishing initial baselines and expectations in the production stage plan	Metrics activity	Analysis
55	_____ is the process of building from a high-level model or concept to build in complexities and lower-level details.	Forward engineering	Software restructuring
56	_____ is nothing but the re-implementation of the legacy system to achieve more sustainability.	Software re-engineering	Software restructuring
57	_____ consumes more time as compared to the reverse engineering.	Forward engineering	Software restructuring

58	_____ is a rebuilding activity	Reengineering	Software restructuring
59	_____dividing project work into short sprints, using adaptive planning and continual improvement, and fostering teams' self-organization and collaboration targeted to produce maximum value.	Agile Project Management	PERT
60	This technique involves detecting the longest path (sequence of tasks) from the beginning to the end of a project and defining the critical tasks.	Agile Project Management	PERT