KARPAGAM ACADEMY OF HIGHER EDUCATION

(Deemed to be University)
(Established Under Section 3 of UGC Act 1956)
Coimbatore - 641021.

(For the candidates admitted from 2016 onwards)

DEPARTMENT OF COMPUTER SCIENCE, CA & IT

SUBJECT NAME: SOFTWARE ENGINEERING **SUB.CODE: 16CSU402**

SEMESTER: IV **CLASS: II B.Sc (CS)**

COURSE OBJECTIVE:

The graduates of the software engineering program shall be able to apply proper theoretical, technical, and practical knowledge of software requirements, analysis, design, implementation, verification and validation, and documentation. This course enables the students to resolve conflicting project objectives considering viable tradeoffs within limitations of cost, time, knowledge, existing systems, and organizations.

COURSE OUTCOME:

- Apply their knowledge of mathematics, sciences, and computer science to the modeling, analysis, and measurement of software artifacts.
- Work effectively as leader/member of a development team to deliver quality software artifacts.
- Analyze, specify and document software requirements for a software system.
- Implement a given software design using sound development practices.
- Verify, validate, assess and assure the quality of software artifacts.
- Design, select and apply the most appropriate software engineering process for a given project, plan for a software project, identify its scope and risks, and estimate its cost and time.
- Express and understand the importance of negotiation, effective work habits, leadership, and good communication with stakeholders, in written and oral forms, in a typical software development environment.

UNIT-I

Introduction: The Evolving Role of Software, Software Characteristics, Changing Nature of Software, Software Engineering as a Layered Technology, Software Process Framework, Framework and Umbrella Activities, Process Models, Capability Maturity Model Integration (CMMI).

UNIT-II:

Requirement Analysis; Initiating Requirement Engineering Process- Requirement Analysis and Modeling Techniques- Flow Oriented Modeling- Need for SRS- Characteristics and Components of SRS- Software Project Management: Estimation in Project Planning Process, Project Scheduling.

UNIT-III:

Risk Management: Software Risks, Risk Identification Risk Projection and Risk Refinement, RMMM plan, **Quality Management-** Quality Concepts, Software Quality Assurance, Software Reviews, Metrics for Process and Projects

UNIT-IV:

Design Engineering-Design Concepts, Architectural Design Elements, Software Architecture, Data Design at the Architectural Level and Component Level, Mapping of Data Flow into Software Architecture, Modeling Component Level Design

UNIT-V

Testing Strategies & Tactics: Software Testing Fundamentals, Strategic Approach to Software Testing, Test Strategies for Conventional Software, Validation Testing, System testing Black-Box Testing, White-Box Testing and their type, Basis Path Testing

Suggested Readings

1. Pressman, R.S. (2009). Software Engineering: A Practitioner‘s Approach (7th ed.). New Delhi: McGraw-Hill.
2. Jalote, P. An Integrated Approach to Software Engineering (2nd ed.). New Delhi: New Age International Publishers.
3. Aggarwal, K.K., & Singh, Y. (2008). Software Engineering (2nd ed.). New Delhi: New Age International Publishers.
4. Sommerville, I. (2006). Software Engineering (8th ed.). New Delhi: Addison Wesley.
5. Bell, D. (2005). Software Engineering for Students (4th ed.) New Delhi: Addison-Wesley.
6. Mall, R. (2004). Fundamentals of Software Engineering (2nd ed.). New Delhi: Prentice-Hall of India.

WEB SITES

1. http://en.wikipedia.org/wiki/Software_engineering
2. http://www.onesmartclick.com/engineering/software-engineering.html
3. http://www.CSU.gatech.edu/classes/AY2000/cs3802_fall/

ESE MARK ALLOCATION

1	Section –A 20 * 1 = 20	20
2	Section – B 5*2 = 10	10
3	Section – C 5*6 = 30	30
	Total	60

KARPAGAM ACADEMY OF HIGHER EDUCATION

(Deemed to be University)
(Established Under Section 3 of UGC Act 1956)
Coimbatore - 641021.

(For the candidates admitted from 2016 onwards)

DEPARTMENT OF COMPUTER SCIENCE, CA & IT

LECTURE PLAN

STAFF NAME: D.SAMPATHKUMAR & S.JOYCE

SUBJECT NAME: SOFTWARE ENGINEERING **SUB.CODE: 16CSU402**

SEMESTER: IV **CLASS: II B.Sc (CS)**

S.No.	Lecture Duration	Topics to be Covered	Support Materials
Unit - I			
1.	1	Introduction to Software Engineering	T1-12
2.	1	The Evolving Role of Software	T1-3
3.	1	Software Characteristics	T1-4
4.	1	Changing Nature of Software	T1-9
5.	1	A Generic View of process	T1-31
6.	1	Software Engineering, A Layered Technology	T1-33
7.	1	Software Framework	T1-33
8.	1	Framework	T1- 14
9.	1	Umbrella Activities	T1- 16
10.	1	Evolutionary Process, Concurrent Models	T1- 43
11.	1	Process Models: Prescriptive Models	T1-38
12	1	Capability Maturity Model Integration (CMMI)	T1-50
		Total No. of Hours Planned for Unit-I	**12**
Textbook		T1: Roger S. Pressman. 2010. Software Engineering – A Practitioner's Approach, 7th Edition, McGraw Hill International Edition, New Delhi.	

Unit – II

1.	1	**Requirement Analysis;**	T1-149
2.	1	Initiating Requirement Engineering Process	T1-153
3.	1	Requirement Analysis and Modeling Techniques	T1-153
4.	1	Flow Oriented Modeling	T1 -187
5.	1	Need for SRS	T1-190
6.	1	Characteristics and Components of SRS	T1-195
7.	1	Components of SRS	W1

8.	1	Software Project Management	T1-647
9.	1	Software Project Management	T1-647
10.	1	Estimation in Project Planning Process	T1-693
11.	1	Project Scheduling.	T1-721
12.	1	Recapitulation and Discussion of important questions	
Total No. of Hours Planned for Unit-I			**12**
Textbook	T1: Roger S. Pressman. 2010. Software Engineering – A Practitioner's Approach, 7th Edition, McGraw Hill International Edition, New Delhi.		

Unit – III

1.	1	Risk Management:	T1-744
2.	1	Software Risks	T1-745
3.	1	Risk Identification	T1-747
4.	1	Risk Projection	T1-749
5.	1	Risk Refinement	T1-749
6.	1	RMMM plan	T1-755
7.	1	Quality Management	T1-397
8.	1	Quality Concepts	T1-398
9.	1	Software Quality Assurance	T1-432
10.	1	Software Reviews	T1-416
11.	1	Metrics for Process and Projects	T1-420
12.	1	Recapitulation and Discussion of important questions	
Total No. of Hours Planned for Unit-I			**12**
Textbook	T1: Roger S. Pressman. 2010. Software Engineering – A Practitioner's Approach, 7th Edition, McGraw Hill International Edition, New Delhi.		

Unit – IV

1.	1	Design Engineering	T1-216
2.	1	Design Concepts: Abstraction Architecture ,Patterns, Separation of Concerns , Modularity , Information Hiding	T1-222
3.	1	Design Concepts: Functional Independence ,Refinement Aspects , Refactoring ,Object-Oriented Design Concepts ,Design Classes	T1-222
4.	1	Architectural Design Elements	T1-222
5.	1	Software Architecture	T1-242
6.	1	Data Design at the Architectural Level and Component Level	T1-234
7.	1	Data Design at Component Level	T1-237

8.	1	Mapping of Data Flow into Software Architecture	T1-261
9.	1	Modeling Component Level Design	T1-276
10.	1	Conducting Component-Level Design, Component-Level Design for WebApps	T1-277
11.	1	Designing Traditional Components, Component-Based Development	T1-298
12.	1	Recapitulation and Discussion of important questions	
		Total No. of Hours Planned for Unit-I	**12**
Textbook		T1: Roger S. Pressman. 2010. Software Engineering – A Practitioner's Approach, 7th Edition, McGraw Hill International Edition, New Delhi.	

Unit – V

1.	1	**Testing Strategies & Tactics**	T1-449
2.	1	Software Testing Fundamentals	T1- 482
3.	1	Strategic Approach to Software Testing	T1-450
4.	1	Test Strategies for Conventional Software	T1-456
5.	1	Validation Testing	T1-467
6.	1	System testing Black-Box Testing	T1-470
7.	1	White-Box Testing and their type	T1-485
8.	1	Basis Path Testing	T1-485
9.	1	Recapitulation and Discussion of important questions	
10.	1	Recapitulation and Discussion of previous semester question papers	
11.	1	Recapitulation and Discussion of previous semester question papers	
12.	1	Recapitulation and Discussion of previous semester question papers	
		Total No. of Hours Planned for Unit-5	**12**
Textbook		T1: Roger S. Pressman. 2010. Software Engineering – A Practitioner's Approach, 7th Edition, McGraw Hill International Edition, New Delhi.	

UNIT-I

SYLLABUS

Introduction: The Evolving Role of Software, Software Characteristics, Changing Nature of Software, Software Engineering as a Layered Technology, Software Process Framework, Framework and Umbrella Activities, Process Models, Capability Maturity Model Integration (CMMI).

Introduction to Software Engineering:

What is software engineering?

Software has become critical to advancement in almost all areas of human Endeavour. The art of programming only is no longer sufficient to construct large programs. There are serious problems in the cost, timeliness, maintenance and quality of many software products. Software engineering has the objective of solving these problems by producing good quality, maintainable software, on time, within budget. To achieve this objective, we have to focus in a disciplined manner on both the quality of the product and on the process used to develop the product.

Definition

At the first conference on software engineering in 1968, Fritz Bauer [FRIT68] defined software engineering as *"The establishment and use of sound engineering principles in order to obtain economically developed software that is reliable and works efficiently on real machines"*. Stephen Schacht [SCHA90] defined the same as *"A discipline whose aim is the production of quality software, software that is delivered on time, within budget, and that satisfies its requirements"*. Both the definitions are popular and acceptable to majority. However, due to increase in cost of maintaining software, objective is now shifting to produce quality software that is maintainable, delivered on time, within budget, and also satisfies its requirements.

The Evolving Role of Software

Software takes on a dual role. It is a product and, at the same time, the vehicle for delivering a product. As a product, it delivers the computing potential embodied by computer hardware or, more broadly, a network of computers that are accessible by local hardware. Whether it resides within a cellular phone or operates inside a mainframe computer, software is information transformer— producing, managing, acquiring, modifying, displaying, or transmitting information that can be as simple as a single bit or as complex as a multimedia presentation. As the vehicle used to deliver the product, software acts as the basis for the control of the computer (operating systems), the communication of information (networks), and the creation and control of other programs (software tools and environments). Software delivers the most important product of our time—information.

Software transforms personal data (e.g., an individual's financial transactions) so that the data can be more useful in a local context; it manages business information to enhance competitiveness; it provides a gateway to worldwide information networks (e.g., Internet) and provides the means for acquiring information in all of its forms.

The role of computer software has undergone significant change over a time span of little more than 50 years. Dramatic improvements in hardware performance, profound changes in computing architectures, vast increases in memory and storage capacity, and a wide variety of exotic input and output options have all precipitated more sophisticated and complex computer-based systems.

The lone programmer of an earlier era has been replaced by a team of software specialists, each focusing on one part of the technology required to deliver a complex application.

Software

Computer software, or just **software**, is a collection of computer programs and related data that provide the instructions for telling a computer what to do and how to do it. In other words, software is a conceptual entity which is a set of computer programs, procedures, and associated documentation concerned with the operation of a data processing system. We can also say software refers to one or more computer programs and data held in the storage of the computer for some purposes.

In other words software is a set of **programs, procedures, algorithms** and its **documentation**. Program software performs the function of the program it implements, either by directly providing instructions to the computer hardware or by serving as input to another piece of software.

The term was coined to contrast to the old term hardware (meaning physical devices). In contrast to hardware, software is intangible, meaning it "cannot be touched". Software is also sometimes used in a more narrow sense, meaning application software only. Sometimes the term includes data that has not traditionally been associated with computers, such as film, tapes, and records.

Software Characteristics

1. **Software is developed or engineered; it is not manufactured in the classical sense.**

Although some similarities exist between software development and hardware manufacture, the two activities are fundamentally different. In both activities, high quality is achieved through good design, but the manufacturing phase for hardware can introduce quality problems that are nonexistent (or easily corrected) for software.

Both activities are dependent on people, but the relationship between people applied

and work accomplished is entirely different . Both activities require the construction of a "product" but the approaches are different. Software costs are concentrated in engineering. This means that software projects cannot be managed as if they were manufacturing projects

2. Software doesn't "wear out."

Fig 1.1 depicts failure rate as a function of time for hardware.

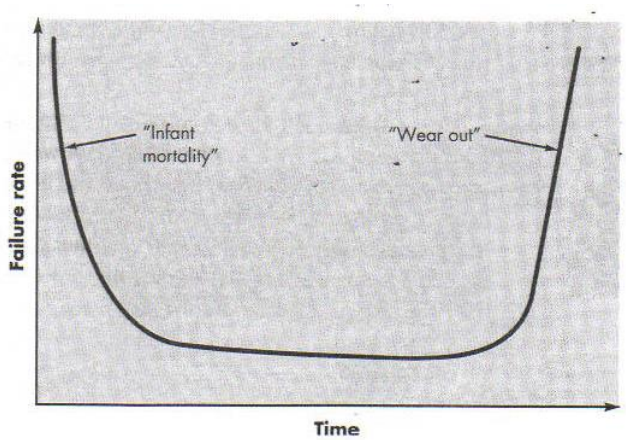


Fig 1.1 Failure curve for hardware

The relationship, often called the "bathtub curve," indicates that hardware exhibits relatively high failure rates early in its life (these failures are often attributable to design or manufacturing defects); defects are corrected and the failure rate drops to a steady-state level (ideally, quite low) for some period of time. As time passes, however, the failure rate rises again as hardware components suffer from the cumulative affects of dust, vibration, abuse, temperature extremes, and many other environmental maladies. Stated simply, the hardware begins to wear out.

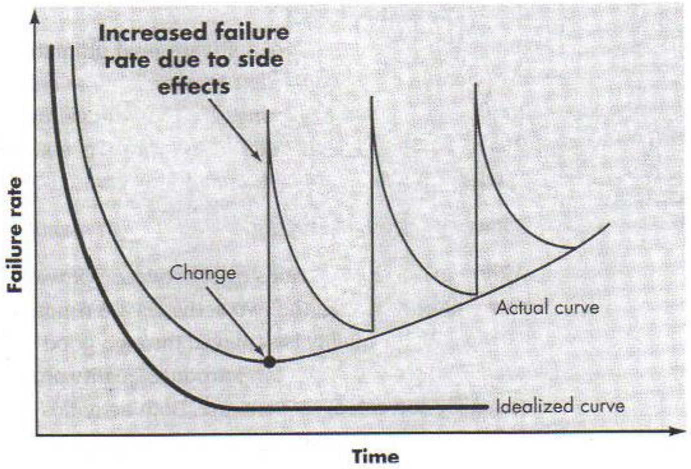


Fig 1.2 Failure curves for software

Software is not susceptible to the environmental maladies that cause hardware to wear out. In theory, therefore, the failure rate curve for software should take the form of the "idealized curve" shown in Fig 1.2. Undiscovered defects will cause high failure rates early in the life of a program. However, these are corrected (ideally, without introducing other errors) and the curve flattens as shown. The idealized curve is a gross oversimplification of actual failure models for software. However, the implication is clear—software doesn't wear out.

This seeming contradiction can best be explained by considering the "actual curve" shown in Fig 1.2. During its life, software will undergo change (maintenance). As changes are made, it is likely that some new defects will be introduced, causing the failure rate curve to spike as shown in Fig 1.2. Before the curve can return to the original steady-state failure rate, another change is requested, causing the curve to spike again. Slowly, the minimum failure rate level begins to rise—the software is deteriorating due to change.

Another aspect of wear illustrates the difference between hardware and software. When a hardware component wears out, it is replaced by a spare part. There are no software spare parts. Every software failure indicates an error in design or in the process through which design was translated into machine executable code. Therefore, software maintenance involves considerably more complexity than hardware maintenance.

3. Although the industry is moving toward component-based assembly, most software continues to be custom built.

Consider the manner in which the control hardware for a computer-based product is designed and built. The design engineer draws a simple schematic of the digital circuitry, does some fundamental analysis to assure that proper function will be achieved, and then goes to the shelf where catalogs of digital components exist. Each integrated circuit (called an *IC* or a *chip*) has a part number, a defined and validated function, a well-defined interface, and a standard set of integration guidelines. After each component is selected, it can be ordered off the shelf.

A software component should be designed and implemented so that it can be reused in many different programs. Modern reusable components encapsulate both data and the processing applied to the data, enabling the software engineer to create new applications from reusable parts. For example, today's graphical user interfaces are built using reusable components that enable the creation of graphics windows, pull-down menus, and a wide variety of interaction mechanisms. The data structure and processing detail required to build the interface are contained with a library of reusable components for interface construction

Changing Nature of software

Four broad categories of software are evolving to dominate the industry.

WebApps

Web-based systems and applications5 (we refer to these collectively as WebApps) were born. WebApps have evolved into sophisticated computing tools that not only provide stand-alone function to the end user, but also have been integrated with corporate databases and business application.

WebApps "involved a mixture between print publishing and software development, between marketing and computing, between internal communications and external relations, and between art and technology."

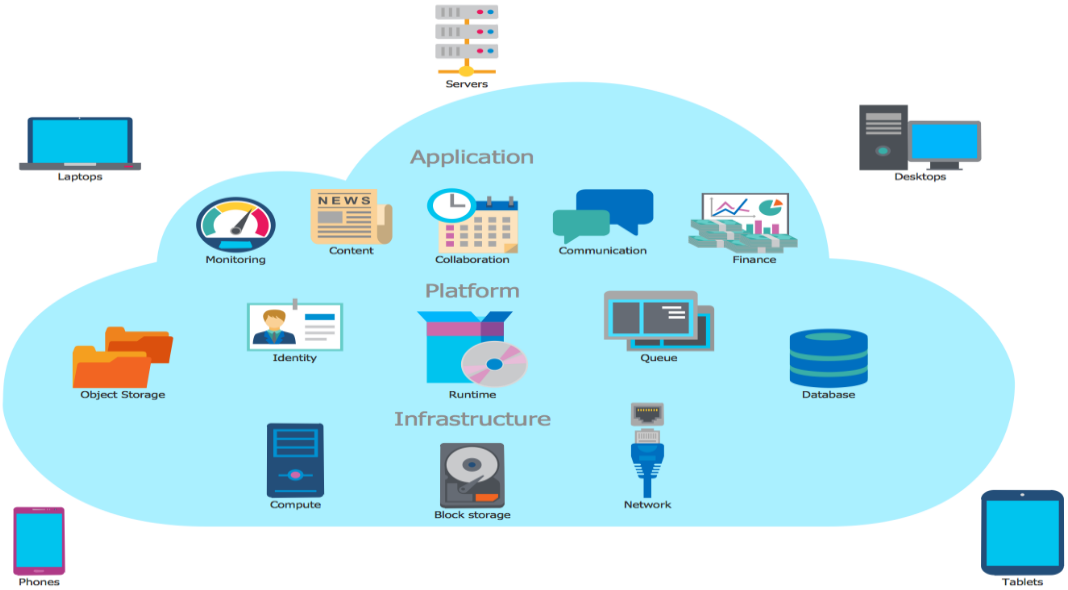
Semantic Web technologies (often referred to as Web 3.0) have evolved into sophisticated corporate and consumer applications that encompass "semantic databases provide new functionality that requires Web linking, flexible data representation, and external access APIs."

Mobile Applications

The term app has evolved to software that has been specifically designed to reside on a mobile platform (e.g., iOS, Android, or Windows Mobile).Mobile applications encompass a user interface that takes advantage of the unique interaction mechanisms provided by the mobile platform, interoperability with Web-based resources that provide access to a wide array of information that is relevant to the app, and local processing capabilities that collect, analyze, and format information in a manner that is best suited to the mobile platform. In addition, a mobile app provides persistent storage capabilities within the platform.

Cloud Computing

Cloud computing encompasses an infrastructure or "ecosystem" that enables any user, anywhere, to use a computing device to share computing resources on a broad scale. The overall logical architecture of cloud computing is represented in Figure



Computing devices reside outside the cloud and have access to a variety of resources within the cloud. These resources encompass applications, platforms, and infrastructure. In its simplest form, an external computing device accesses the cloud via a Web browser or analogous software. The cloud provides access to data that resides with databases and other data structures. The implementation of cloud computing requires the development of an architecture that encompasses front-end and back-end services. The front-end includes the client (user) device and the application software (e.g., a browser) that allows the back-end to be accessed. The back-end includes servers and related computing resources, data storage systems (e.g., databases).

Product Line Software

The Software Engineering Institute defines a software product line as "a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission. A software product line shares a set of assets that include requirements, architecture, design patterns, reusable components, test cases, and other software engineering work products. A software product line results in the development of many products that are engineered by capitalizing on the commonality among all the products within the product line.

A Generic View of process

Software Engineering as a Layered Technology

Any engineering approach much rests on organizational approach to quality, e.g. total quality management and such emphasize continuous process improvement (that is increasingly more effective approaches to software engineering). The bedrock that supports a software engineering is a **quality focus**.

The foundation for software engineering is the **process** layer. Software engineering process is the glue that holds the technology layers together and enables rational and timely development of computer software. Process defines a framework for a set of key process areas (KPAs) that must be established for effective delivery of software engineering technology. The key process areas form the basis for management

control of software projects and establish the context in which technical methods are applied, work products (models, documents, data, reports, forms, etc.) are produced, milestones are established, quality is ensured, and change is properly managed.

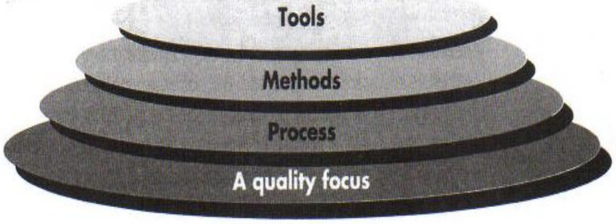


Fig 1.3 Software Engineering Layers

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

Software engineering **tools** provide automated or semi-automated support for the process and the methods. When tools are integrated so that information created by one tool can be used by another, a system for the support of software development, called computer-aided software engineering, is established.

Process Framework

Identifies a small number of framework activities that are applicable to all software projects. In addition the framework encompasses umbrella activities that are applicable across the software process.

Generic Process Framework Activities

Each framework activity is populated by a set of software engineering actions. An action, e.g. design, is a collection of related tasks that produce a major software engineering work product.

Communication – lots of communication and collaboration with customer and other stakeholders. Encompasses requirement gathering.

Planning – establishes plan for software engineering work that follows. Describes technical tasks, likely risks, required resources, works products and a work schedule

Modeling – encompasses creation of models that allow the developer and customer to better understand software requirements and the design that will achieve those requirements.

Modeling Activity – composed of two software engineering actions

• **analysis** – composed of work tasks (e.g. requirement gathering, elaboration, specification and validation) that lead to creation of analysis model and/or requirements specification.

• design – encompasses work tasks such as data design, architectural design, interface design and component level design leads to creation of design model and/or a design specification.

Construction – code generation and testing.

Deployment – software, partial or complete, is delivered to the customer who evaluates it and provides feedback. Different projects demand different task sets. Software team chooses task set based on problem and project characteristics.

THE SOFTWARE PROCESS

A process is a collection of activities, actions, and tasks that are performed when some work product is to be created. An activity strives to achieve a broad objective (e.g., communication with stakeholders). An action (e.g., architectural design) encompasses a set of tasks that produce a major work product (e.g., an architectural model). A task focuses on a small, but well-defined objective (e.g., conducting a unit test) that produces a tangible outcome.

The Process Framework

A process framework establishes the foundation for a complete software engineering process by identifying a small number of framework activities that are applicable to all software projects. In addition, the process framework encompasses a set of umbrella activities that are applicable across the entire software process. A generic process framework for software engineering encompasses five activities:

1. **Communication.** Before any technical work can commence, it is critically important to communicate and collaborate with the customer (and other stakeholders). The intent is to understand stakeholders' objectives for the project and to gather requirements that help define software features and functions.
2. **Planning.** A software project is a complicated journey, and the planning activity creates a "map" that helps guide the team as it makes the journey. The map—called a software

project plan—defines the software engineering work by describing the technical tasks to be conducted, the risks that are likely, the resources that will be required, the work products to be produced, and a work schedule.

3. **Modeling.** Whether you're a landscaper, a bridge builder, an aeronautical engineer, a carpenter, or an architect, you work with models. You create a "sketch" of the thing so that you'll understand the big picture—what it will look like architecturally, how the constituent parts fit together, and many other characteristics. A software engineer does the same thing by creating models to better understand software requirements and the design that will achieve those requirements.
4. **Construction.** What you design must be built. This activity combines code generation (either manual or automated) and the testing that is required to uncover errors in the code.
5. **Deployment.** The software (as a complete entity or as a partially completed increment) is delivered to the customer who evaluates the delivered product and provides feedback based on the evaluation.

Umbrella Activities

Software engineering process framework activities are complemented by a number of umbrella activities. In general, umbrella activities are applied throughout a software project and help a software team manage and control progress, quality, change, and risk. Typical umbrella activities include:

Software project tracking and control —allows the software team to assess progress against the project plan and take any necessary action to maintain the schedule.

Risk management —assesses risks that may affect the outcome of the projector the quality of the product.

Software quality assurance —defines and conducts the activities required to ensure software quality.

Technical reviews —assess software engineering work products in an effort to uncover and remove errors before they are propagated to the next activity.

Measurement —defines and collects process, project, and product measures that assist the team in delivering software that meets stakeholders' needs; can be used in conjunction with all other framework and umbrella activities.

Software configuration management —manages the effects of change throughout the software process.

Reusability management —defines criteria for work product reuse (including software components) and establishes mechanisms to achieve reusable components.

Work product preparation and production —encompass the activities required to create work products such as models, documents, logs, forms, and lists.

PROCESS MODELS

A process model provides a specific roadmap for software engineering work. It defines the flow of all activities, actions and tasks, the degree of iteration, the work products, and the organization of the work that must be done. Software engineers and their managers adapt a process model to their needs and then follow it. In addition, the people who have requested the software have a role to play in the process of defining, building, and testing it. The process model provides you with the "steps" you'll need to perform disciplined software engineering work. From the point of view of a software engineer, the work product is a customized description of the activities and tasks defined by the process.

PRESCRIPTIVE PROCESS MODELS

A prescriptive process model strives for structure and order in software development. Activities and tasks occur sequentially with defined guidelines for progress. The prescriptive process approach in which order and project consistency are dominant issues. We call them "prescriptive" because they prescribe a set of process elements—framework activities, software engineering actions, tasks, work products, quality assurance, and change control mechanisms for each project. Each process model also prescribes a process flow (also called a work flow)—that is, the manner in which the process elements are interrelated to one another.

The following framework activities are carried out irrespective of the process model chosen by the organization.

1. Communication
2. Planning
3. Modeling
4. Construction
5. Deployment

The name 'prescriptive' is given because the model prescribes a set of activities, actions, tasks, quality assurance and change the mechanism for every project.

There are three types of prescriptive process models. They are:

1. The Waterfall Model
2. Incremental Process model
3. RAD model

1. The Waterfall Model

- The waterfall model is also called as **'Linear sequential model'** or **'Classic life cycle model'.**
- In this model, each phase is fully completed before the beginning of the next phase.
- This model is used for the small projects.
- In this model, feedback is taken after each phase to ensure that the project is on the right path.
- Testing part starts only after the development is complete.

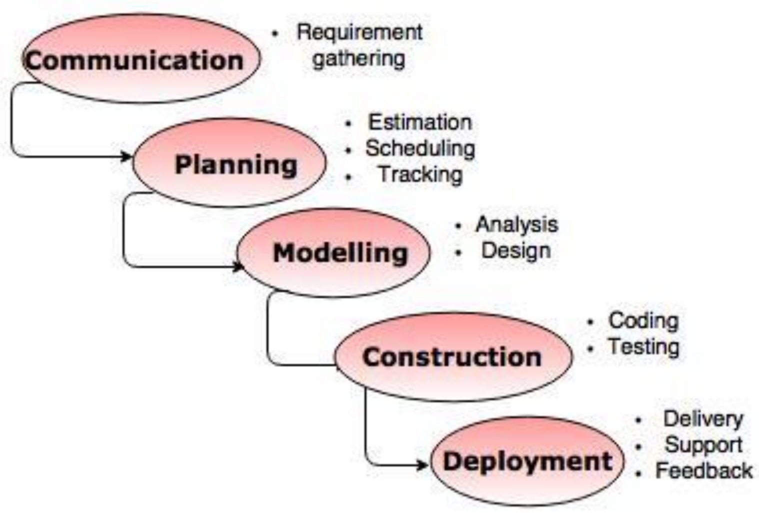


Fig. - The Waterfall model

The description of the phases of the waterfall model is same as that of the process model.

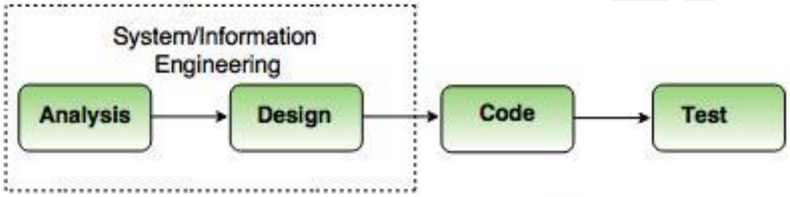


Fig. - The linear sequential model

Advantages of waterfall model

- The waterfall model is simple and easy to understand, implement, and use.
- All the requirements are known at the beginning of the project, hence it is easy to manage.
- It avoids overlapping of phases because each phase is completed at once.
- This model works for small projects because the requirements are understood very well.
- This model is preferred for those projects where the quality is more important as compared to the cost of the project.

Disadvantages of the waterfall model

- This model is not good for complex and object oriented projects.
- It is a poor model for long projects.
- The problems with this model are uncovered, until the software testing.
- The amount of risk is high.

2. Incremental Process model

- The incremental model combines the elements of waterfall model and they are applied in an iterative fashion.
- The first increment in this model is generally a core product.
- Each increment builds the product and submits it to the customer for any suggested modifications.
- The next increment implements on the customer's suggestions and add additional requirements in the previous increment.
- This process is repeated until the product is finished.

For example, the word-processing software is developed using the incremental model.

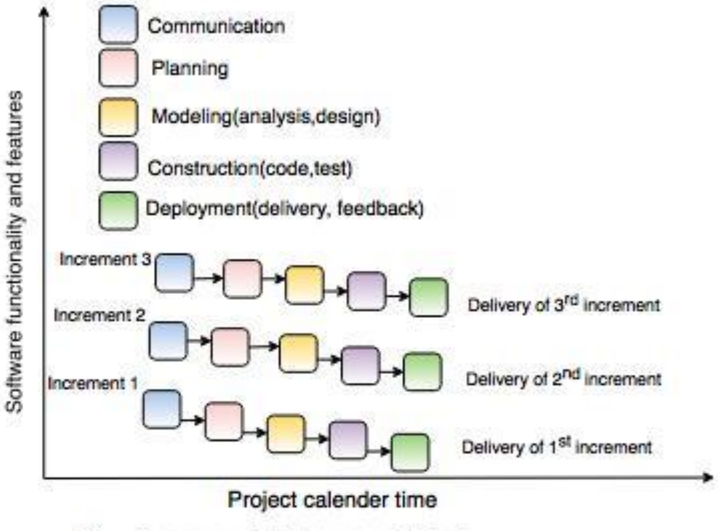

Communication
Planning
Modeling(analysis,design)
Construction(code,test)
Deployment(delivery, feedback)
Increment 3
Delivery of 3rd increment
Increment 2
Delivery of 2nd increment
Increment 1
Delivery of 1st increment
Software functionality and features
Project calender time

Fig. - Incremental Process Model

Advantages of incremental model

- This model is flexible because the cost of development is low and initial product delivery is faster.
- It is easier to test and debug during the smaller iteration.
- The working software generates quickly and early during the software life cycle.
- The customers can respond to its functionalities after every increment.

Disadvantages of the incremental model

- The cost of the final product may cross the cost estimated initially.
- This model requires a very clear and complete planning.
- The planning of design is required before the whole system is broken into small increments.
- The demands of customer for the additional functionalities after every increment causes problem during the system architecture.

3. RAD model

- RAD is a Rapid Application Development model.
- Using the RAD model, software product is developed in a short period of time.
- The initial activity starts with the communication between customer and developer.
- Planning depends upon the initial requirements and then the requirements are divided into groups.
- Planning is more important to work together on different modules.

The RAD model consist of following phases:

1. Business Modeling

- Business modeling consist of the flow of information between various functions in the project.
- For example what type of information is produced by every function and which are the functions to handle that information.

- A complete business analysis should be performed to get the essential business information.

2. Data modeling

- The information in the business modeling phase is refined into the set of objects and it is essential for the business.
- The attributes of each object are identified and define the relationship between objects.

3. Process modeling

- The data objects defined in the data modeling phase are changed to fulfil the information flow to implement the business model.
- The process description is created for adding, modifying, deleting or retrieving a data object.

4. Application generation

- In the application generation phase, the actual system is built.
- To construct the software the automated tools are used.

5. Testing and turnover

- The prototypes are independently tested after each iteration so that the overall testing time is reduced.
- The data flow and the interfaces between all the components are fully tested. Hence, most of the programming components are already tested.

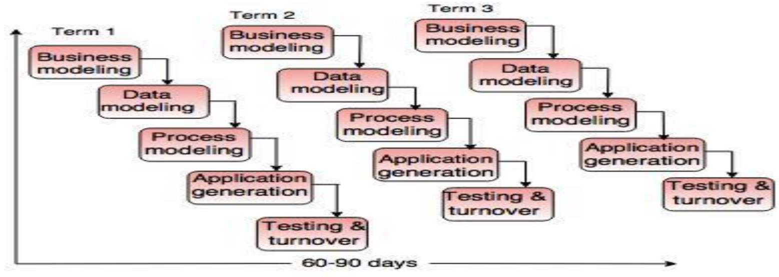


Fig. - RAD Model

Evolutionary Process Models

- Evolutionary models are iterative type models.

- They allow to develop more complete versions of the software.

Following are the evolutionary process models.

1. The prototyping model
2. The spiral model
3. Concurrent development model

1. The Prototyping model

- Prototype is defined as first or preliminary form using which other forms are copied or derived.
- Prototype model is a set of general objectives for software.
- It does not identify the requirements like detailed input, output.
- It is software working model of limited functionality.
- In this model, working programs are quickly produced.

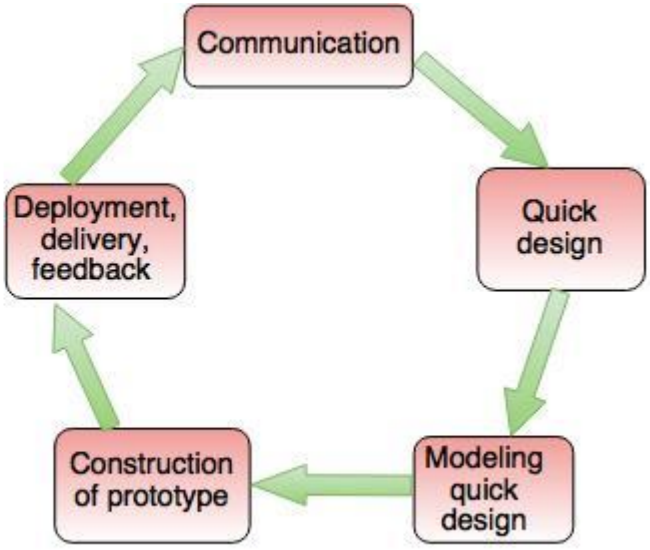


Fig. - The Prototyping Model

The different phases of Prototyping model are:

1. Communication

In this phase, developer and customer meet and discuss the overall objectives of the software.

2. Quick design

- Quick design is implemented when requirements are known.
- It includes only the important aspects like input and output format of the software.
- It focuses on those aspects which are visible to the user rather than the detailed plan.
- It helps to construct a prototype.

3. Modeling quick design

- This phase gives the clear idea about the development of software because the software is now built.
- It allows the developer to better understand the exact requirements.

4. Construction of prototype

The prototype is evaluated by the customer itself.

5. Deployment, delivery, feedback

- If the user is not satisfied with current prototype then it refines according to the requirements of the user.
- The process of refining the prototype is repeated until all the requirements of users are met.
- When the users are satisfied with the developed prototype then the system is developed on the basis of final prototype.

Advantages of Prototyping Model

- Prototype model need not know the detailed input, output, processes, adaptability of operating system and full machine interaction.
- In the development process of this model users are actively involved.
- The development process is the best platform to understand the system by the user.
- Errors are detected much earlier.

- Gives quick user feedback for better solutions.
- It identifies the missing functionality easily. It also identifies the confusing or difficult functions.

Disadvantages of Prototyping Model:

- The client involvement is more and it is not always considered by the developer.
- It is a slow process because it takes more time for development.
- Many changes can disturb the rhythm of the development team.
- It is a thrown away prototype when the users are confused with it.

2. The Spiral model

- Spiral model is a risk driven process model.
- It is used for generating the software projects.
- In spiral model, an alternate solution is provided if the risk is found in the risk analysis, then alternate solutions are suggested and implemented.
- It is a combination of prototype and sequential model or waterfall model.
- In one iteration all activities are done, for large project's the output is small.

The framework activities of the spiral model are as shown in the following figure.

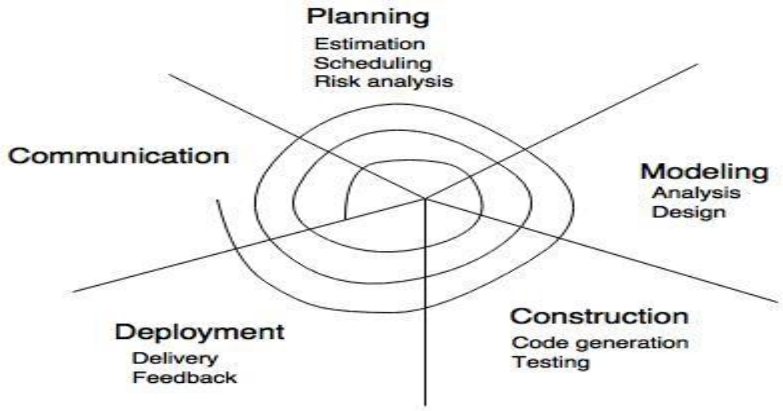


Fig. - The Spiral Model

The description of the phases of the spiral model is same as that of the process model.

Advantages of Spiral Model

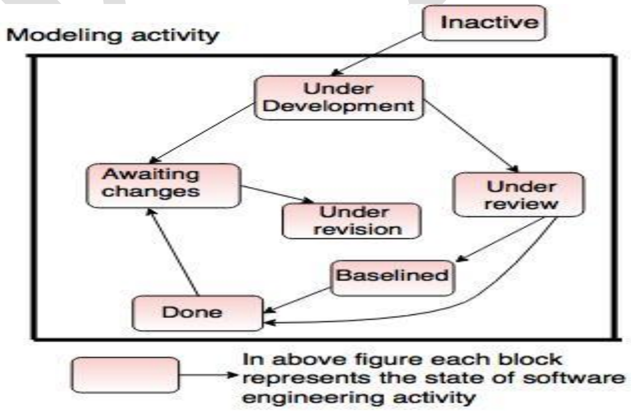
- It reduces high amount of risk.
- It is good for large and critical projects.
- It gives strong approval and documentation control.
- In spiral model, the software is produced early in the life cycle process.

Disadvantages of Spiral Model

- It can be costly to develop a software model.
- It is not used for small projects.

3. The concurrent development model

- The concurrent development model is called as concurrent model.
- The communication activity has completed in the first iteration and exits in the awaiting changes state.
- The modeling activity completed its initial communication and then go to the underdevelopment state.
- If the customer specifies the change in the requirement, then the modeling activity moves from the under development state into the awaiting change state.
- The concurrent process model activities moving from one state to another state.



- **Fig. - One element of the concurrent process model**

Advantages of the concurrent development model

- This model is applicable to all types of software development processes.
- It is easy for understanding and use.
- It gives immediate feedback from testing.
- It provides an accurate picture of the current state of a project.

Disadvantages of the concurrent development model

- It needs better communication between the team members. This may not be achieved all the time.
- It requires to remember the status of the different activities.

SPECIALIZED PROCESS MODELS

1. **Component Based Development**

Commercial off-the-shelf (COTS) Software components, developed by vendors who offer them as products, can be used when Software is to be built. These components provide targeted functionality with well-defined interfaces that enable the component to be integrated into the Software.

The component-based development model incorporates many of the characteristics of the spiral model.

The component-based development model incorporates the following steps:

- Available component-based products are researched and evaluated for the application domain in question.
- Component integration issues are considered.
- Software architecture is designed to accommodate the components.
- Components are integrated into the architecture.
- 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.

2. The Formal Methods Model

The Formal Methods Model encompasses a set of activities that leads to formal mathematical specifications of Software.

Formal methods enable a Software engineer to specify, develop, and verify a computer-based system by applying a rigorous, mathematical notation.

A variation of this approach, called clean-room Software engineering is currently applied by some software development organizations.

Although not a mainstream approach, the formal methods model offers the promise of defect-free Software. Yet, concern about its applicability in a business environment has been voiced:

- The development of formal models is currently quite time-consuming and expensive.
- B/C few software developers have the necessary background to apply formal methods, extensive training is required.
- It is difficult to use the methods as a communication mechanism for technically unsophisticated customers.

3. Aspect-Oriented Software Development

Regardless of the software process that is chosen, the builders of complex software invariably implement a set of localized features, functions, and information content.

CAPABILITY MATURITY MODEL INTEGRATION (CMMI)

The Capability Maturity Model Integration (CMMI) is a capability maturity model developed by the Software Engineering Institute, part of Carnegie Mellon University in Pittsburgh, USA. The CMMI principal is that “the quality of a system or product is highly influenced by the process used to develop and maintain it”. CMMI can be used to guide process improvement across a project, a division, or an entire organization

CMMI provides:

Guidelines for processes improvement

An integrated approach to process improvement

Embedding process improvements into a state of business as usual

A phased approach to introducing improvements

CMMI Models

CMMI consists of three overlapping disciplines (constellations) providing specific focus into the Development, Acquisition and Service Management domains respectively:

CMMI for Development (CMMI-DEV) – Product and service development

CMMI for Services (CMMI-SVC) – Service establishment, management, and delivery

CMMI for Acquisition (CMMI-ACQ) – Product and service acquisition

CMMI Maturity Levels

There are five CMMI maturity levels. However, maturity level ratings are only awarded for levels 2 through 5.

CMMI Maturity Level 2 – Managed

CM – Configuration Management

MA – Measurement and Analysis

PMC – Project Monitoring and Control

PP – Project Planning

PPQA – Process and Product Quality Assurance

REQM – Requirements Management

SAM – Supplier Agreement Management

CMMI Maturity Level 3 – Defined

DAR – Decision Analysis and Resolution

IPM – Integrated Project Management +IPPD

OPD – Organizational Process Definition +IPPD

OPF – Organizational Process Focus

OT – Organizational Training

PI – Product Integration

RD – Requirements Development

RSKM – Risk Management

TS – Technical Solution

VAL – Validation

VER – Verification

CMMI Maturity Level 4 – Quantitatively Managed

QPM – Quantitative Project Management

OPP – Organizational Process Performance

CMMI Maturity Level 5 – Optimizing

CAR – Causal Analysis and Resolution

OID – Organizational Innovation and Deployment

CMMI Appraisals

Certification does not exist as a concept in CMMI, rather an organisation is appraised. and can be awarded a maturity level rating (1-5) or a capability level achievement profile.

Many organizations find value in measuring their progress by conducting an appraisal in order to:

Determine how well the organization's processes compare to CMMI best practices, and to identify areas where improvement can be made Inform external customers and suppliers of how well the organization's processes compare to CMMI best practices

Meet the contractual requirements of one or more customers

Possible Questions

Part – B (2 Mark)

1. Define software engineering.
2. Differentiate between software and hardware characteristics
3. What are the major process framework activities for software engineering?
4. List the major disadvantages of Waterfall model.
5. What is CMMI? List the five maturity levels of CMMI.

Part – C (6 Mark)

1. Explain in detail about Concurrent Development model.
2. Discuss in detail about Spiral Model.
3. Discuss in detail about the umbrella activities in the software process framework.
4. Illustrate Prototyping model with its phases.
5. Explain in detail about Waterfall Model with a neat sketch.
6. Discuss in detail about the Layered perspective of software engineering.
7. Describe in detail the changes acquired in the nature of software.
8. Discuss in detail about the Rapid Application Development Model with a neat sketch.
9. Explain in detail about Incremental Process Model.
10. Explain in detail about Prescriptive model.

Enable | Enlighten | Enrich
KARPAGAM
ACADEMY OF HIGHER EDUCATION
(Deemed to be University)
(Established Under Section 3 of UGC Act, 1956)

KARPAGAM ACADEMY OF HIGHER EDUCATION

DEPARTMENT OF COMPUTER SCIENCE

II BSc cs BATCH(2016 - 2019)

PART-A OBJECTIVE TYPE/ MULTIPLE CHOICE QUESTIONS

UNIT-1

S.NO	Questions	opt1	opt2	opt3	opt4	Answer
1	Software takes on a ______________ role.	single	dual	triple	tetra	dual
2	Software is a _______________.	virtual	system	modifier	framework	modifier
3	Instructions that when executed provide desired function and performance is called	software	hardware	firmware	humanware	software
4	High quality of software is achieved through ________________.	testing	good design	construction	manufacture	good design
5	Software doesn't ________________.	tearout	wearout	degrade	deteriorate	wearout
6	Software is not susceptible to ______________.	hardware	defects	environmental melodies	deterioration	environmental melodies
7	Software will undergo __________.	database	testing	enhancement	manufacture	enhancement
8	_________ refers to the meaning and form of incoming and outgoing information.	content	software	hardware	data	content
9	_____________ refers to the predictability of the order and timing of information.	system software	network software	information determinacy	database	information determinacy
10	____________ is not a system software.	MS Office	compiler	editor	file management utility	MS Office
11	Collection of programs written to service other programs are called __________.	system software	business software	embedded software	pc software	system software
12	Which one is not coming under software myths	Management myths	customer myths	product myths	practitioners myths	product myths
13	_________ is a PC Software.	MS word	LISP	CAD	C	MS word
14	Software that monitors, analyses, controls real world events is called _________.	Business software	real time software	web based software	embedded software	real time software
15	The bedrock that supports software engineering is a______	tools	methods	process models	quality focus	quality focus
16	A complete software process by identifying a small number of _____	framework activities	umbrella activities	process framework	software process	framework activities
17	The process framework encompassess a set of ________	framework activities	umbrella activities	process framework	software process	umbrella activities

18	software engineering action is________________	design	chronic	decision	crisis	design
19	Which one is effect the outcome of the project?	Risk management	Measurement	technical reviews	Reusability	Risk management
20	Continuing indefinitely is called ___________.	crisis	decision	affliction	chronic	chronic
21	Component based development uses ____________.	functions	subroutines	procedures	objects	objects
22	UML stands for ____________.	Universal Modelling Language	User Modified Language	Unified Modelling Language	User Model Language	Unified Modelling Language
23	A model which uses formal mathematical specification is called ________.	4 GT model	Unified method model	formal methods model	component based development	formal methods model
24	A variation of formal methods model is called _____________.	component based development	4 GT model	unified method model	cleanroom software engineering	cleanroom software engineering
25	The development of formal methods is ___________.	less time consuming	quite time consuming	does not consume time	very less time consuming	quite time consuming
26	The first step to develop software is ______________.	analysis	design	requirements gathering	coding	requirements gathering
27	The waterfall model sometimes called as	classic model	classic life cycle model	life cycle model	cycle model	classic life cycle model
28	Software engineering activities include ____________	decision	affliction	hardware	maintenance	maintenance
29	all process model prescribes a ______________.	circular	elliptical	spiral	workflow	workflow
30	Component based development incorporates the characteristics of the ___________ model	circular	elliptical	spiral	hierarchical	spiral
31	Prototype is a ______________.	software	hardware	computer	model	model
32	For small applications it is possible to move from requirement gathering step to____________.	analysis	implementation	design	modeling	implementation
33	Software project management begins with a set of activities that are collectively called ________________	project planning	software scope	software estimation	decomposition	project planning
34	Breaking up of a complex problem into small steps is called ____________.	project planning	software scope	software estimation	decomposition	decomposition
35	The ease with which software can be transferred from one computer to another. This quality attribute is called ______________.	portability	reliability	efficiency	accuracy	portability
36	The ability of a program to perform a required function under stated condition for a stated period of time. This quality attribute is called ____________.	portability	reliability	efficiency	accuracy	reliability
37	The event to which software performs its intended function. This quality attribute is called _________________.	portability	reliability	efficiency	accuracy	efficiency

38	A qualitative assessments of freedom from errors. This quality attribute is called ____________.	portability	reliability	efficiency	accuracy	accuracy
39	The extent to which software can continue to operate correctly. This quality attribute is called ______________.	robustness	correctness	efficiency	reliability	robustness
40	The extent to which the software is free from design and coding defects ie fault free. This quality attribute is called _____________.	robustness	correctness	efficiency	reliability	correctness
41	System shall reside in 50KB of memory is an example of _____________.	quantified requirement	qualified requirement	functional requirement	performance requirement	quantified requirement
42	Accuracy shall be sufficient to support mission is an example of ___________.	quantified requirement	qualified requirement	functional requirement	performance requirement	qualified requirement
43	System shall make efficient use of memory is an example of ______________.	quantified requirement	qualified requirement	functional requirement	performance requirement	qualified requirement
44	Which level of CMM is for process control?	Initial	Repeatable	Defined	Optimizing	Optimizing
45	Product is	Deliverables	User expectations	Organization's effort in development	none of the above	Deliverables
46	To produce a good quality product, process should be	Complex	Efficient	Rigorous	none of the above	Efficient
47	Which is not a product metric?	Size	Reliability	Productivity	Functionality	Productivity
48	Which is NOT a process metric?	Productivity	Functionality	Quality	Efficiency	Functionality
49	Effort is measured in terms of:	Person-months	Rupees	Persons	Months	Person-months
50	An independently deliverable piece of functionality providing access to its services through interface is called	Software measurement	Software composition	Software measure	Software component	Software component
51	Management of software development is dependent on	People	product	Process	all of the above	all of the above
52	During software development, which factor is most crucial?	People	Product	Process	Project	People
53	Program is	Subset of software	super set of software	Software	none of the above	Subset of software
54	Milestones are used to	Know the cost of the project	know the status of the project	Know user expectations	none of the above	know the status of the project
55	Software engineering approach is used to achieve:	Better performance of hardware	Error free software	Reusable software	Quality software product	Quality software product
56	Software consists of	instructions + operating system	documentation + operating procedures	Programs + hardware manuals	Set of programs	documentation + operating procedures
57	CASE Tool is	Aided Software Engineering	Aided Software Engineering	Aided Software Engineering	Analysis Software Engineering	Aided Software Engineering
58	SDLC stands for	Software design life cycle	Software development life cycle	System development life cycle	System design life cycle	Software development life cycle
59	RAD stands for	Rapid application development	Relative application development	Ready application development	Repeated application development	Rapid application development

60	Which phase is not available in software life cycle?	Coding	Testing	Maintenance	Abstraction	Abstraction

UNIT-II

SYLLABUS

Requirement Analysis; Initiating Requirement EngineeringProcess- Requirement Analysis and Modeling Techniques- FlowOriented Modeling- Need for SRS- Characteristics and Components of SRS- Software Project Management: Estimation in Project Planning Process, Project Scheduling.

REQUIREMENT ANALYSIS

Introduction to requirement engineering

- The process of collecting the software requirement from the client then understand, evaluate and document it is called as requirement engineering.
- Requirement engineering constructs a bridge for design and construction.

Requirement engineering consists of seven different tasks as follow:

1. Inception

- Inception is a task where the requirement engineering asks a set of questions to establish a software process.
- In this task, it understands the problem and evaluates with the proper solution.
- It collaborates with the relationship between the customer and the developer.
- The developer and customer decide the overall scope and the nature of the question.

2. Elicitation

Elicitation means to find the requirements from anybody.

The requirements are difficult because the **following problems occur in elicitation**.

Problem of scope: The customer give the unnecessary technical detail rather than clarity of the overall system objective.

Problem of understanding: Poor understanding between the customer and the developer regarding various aspect of the project like capability, limitation of the computing environment.

Problem of volatility: In this problem, the requirements change from time to time and it is difficult while developing the project.

3. Elaboration

- In this task, the information taken from user during inception and elaboration and are expanded and refined in elaboration.
- Its main task is developing pure model of software using functions, feature and constraints of a software.

4. Negotiation

In negotiation task, a software engineer decides the how will the project be achieved with limited business resources.

- To create rough guesses of development and access the impact of the requirement on the project cost and delivery time.

5. Specification

- In this task, the requirement engineer constructs a final work product.
- The work product is in the form of software requirement specification.
- In this task, formalize the requirement of the proposed software such as informative, functional and behavioral.
- The requirement are formalize in both graphical and textual formats.

6. Validation

- The work product is built as an output of the requirement engineering and that is accessed for the quality through a validation step.
- The formal technical reviews from the software engineer, customer and other stakeholders helps for the primary requirements validation mechanism.

7. Requirement management

- It is a set of activities that help the project team to identify, control and track the requirements and changes can be made to the requirements at any time of the ongoing project.

- These tasks start with the identification and assign a unique identifier to each of the requirement.
- After finalizing the requirement traceability table is developed.
- The examples of traceability table are the features, sources, dependencies, subsystems and interface of the requirement

Initiating requirement engineering process

Eliciting Requirements

Eliciting requirement helps the user for collecting the requirement

Eliciting requirement steps are as follows:

1. Collaborative requirements gathering

- Gathering the requirements by conducting the meetings between developer and customer.
- Fix the rules for preparation and participation.
- The main motive is to identify the problem, give the solutions for the elements, negotiate the different approaches and specify the primary set of solution requirements in an environment which is valuable for achieving goal.

2. Quality Function Deployment (QFD)

- In this technique, translate the customer need into the technical requirement for the software.
- QFD system designs a software according to the demands of the customer.

QFD consist of three types of requirement:

Normal requirements

- The objective and goal are stated for the system through the meetings with the customer.
- For the customer satisfaction these requirements should be there.

Expected requirement

- These requirements are implicit.
- These are the basic requirement that not be clearly told by the customer, but also the customer expect that requirement.

Exciting requirement

- These features are beyond the expectation of the customer.
- The developer adds some additional features or unexpected feature into the software to make the customer more satisfied.

For example, the mobile phone with standard features, but the developer adds few additional functionalities like voice searching, multi-touch screen etc. then the customer more exited about that feature.

3. Usage scenarios

- Till the software team does not understand how the features and function are used by the end users it is difficult to move technical activities.
- To achieve above problem the software team produces a set of structure that identify the usage for the software.
- This structure is called as 'Use Cases'.

4. Elicitation work product

- The work product created as a result of requirement elicitation that is depending on the size of the system or product to be built.
- The work product consists of a statement need, feasibility, and statement scope for the system.
- It also consists of a list of users participate in the requirement elicitation.
- Analysis model operates as a link between the 'system description' and the 'design model'.
- In the analysis model, information, functions and the behavior of the system is defined and these are translated into the architecture, interface and component level design in the 'design modeling'.

Building the Analysis model

Requirement Analysis

Requirement Analysis results in the specification of software's operational characteristics indicates software interface with other system elements and establishes constraints that software must meet

Requirement analysis allow the software engineer to elaborate on basic requirements established during earlier requirement engineering tasks and build models that depict user scenario, functional activities, problem classes and their relationships, system and class behavior, and the flow of data as it is transformed.

Requirement analysis provides the software designer with a representation of information, function and behavior that can be translated to architectural, interface and component-level designs

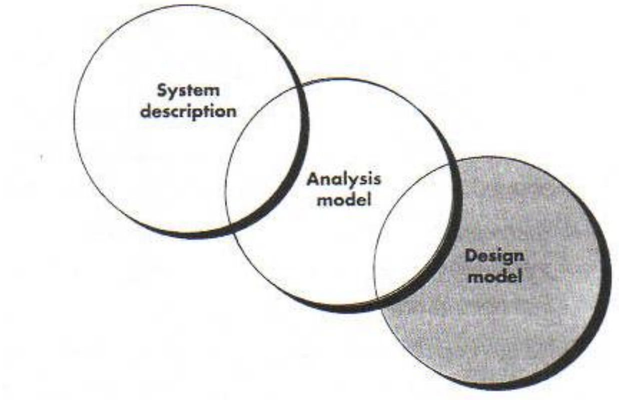
Finally, the analysis model and the requirement specification provide the developer and customer with the means to assess quality once software is built. Throughout analysis modeling, the software engineer's primary focus is on what and not how

1. Overall Objectives and Philosophy

The analysis model must have three primary objectives

- To describe what the customer requires
- To establish a basis for the creation of software design
- To define a set of requirements that can be validated once the software is built

The analysis model bridges the gap between a system level description that describes overall system functionality as it is achieved by applying software, hardware, data, human and other system elements and a software design that describes the software's application architecture, user interface and component level structure

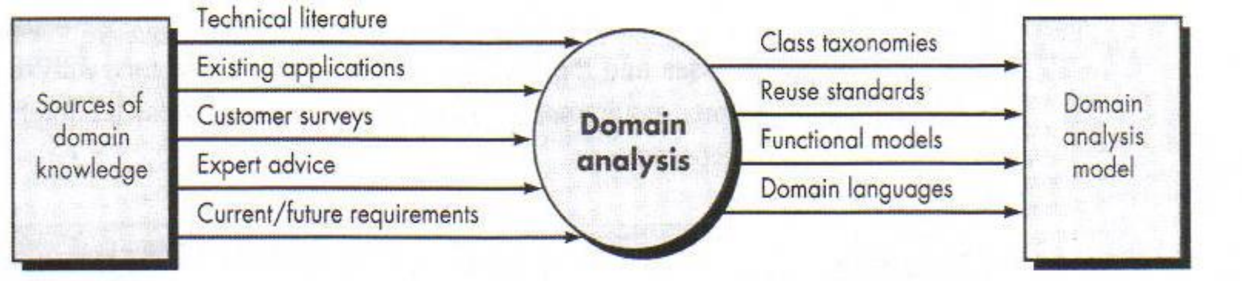


2. Analysis rules of Thumb

- The model should focus on requirements that are visible within the problem or business domain. The level of abstraction should be relatively high
- Each element of the analysis model should add to an overall understanding of software requirements and provide insight into the information domain, function and behavior of the system
- Delay consideration of infrastructure and non functional models until design
- Minimize coupling throughout the system
- Be certain that the analysis model provides value to all stakeholders
- Keep the model as simple as it can be

3. Domain Analysis

The analysis patterns often reoccur across many applications within a specific business domain. If these patterns are defined and categorized in a manner that allows a software engineer or analyst to recognize and reuse them, the creation of the analysis model is expedited.



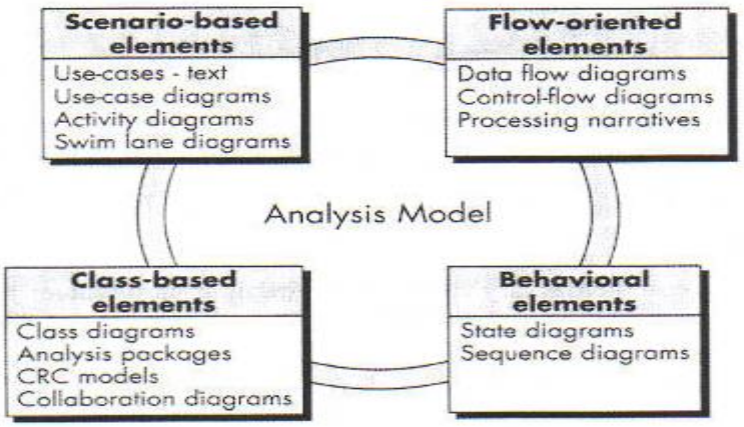
Input and output of Domain Analysis

Software domain analysis is the identification, analysis, and specification of common requirements from a specific application domain, typically for reuse on multiple projects within the application domain.

Analysis modeling approaches

One view of analysis modeling, called structured analysis, considers Data and the processes that transform the dada as separate entities. Data objects are modeled in a way that defines their attributes and relationships. Processes that manipulate data objects are modeled in a manner that shows how they transform data as a data flow through the system.

A second approach to analysis modeling, called objects oriented analysis, focuses on the definition of classes and the manner in which they collaborate with one another to effect customer requirements.



Data modeling concepts

Data Modeling Concepts

Analysis modeling often begins with data modeling. The 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.

1. Data object

A data object is a representation of almost any composite information that must be understood by software. By composite information, we 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.

A data object can be an external entity (e.g., anything that produces or consumes information), a thing (e.g., a report or a display), an occurrence (e.g., a telephone call)

or event (e.g., an alarm), a role (e.g., salesperson), an organizational unit (e.g., accounting department), a place (e.g., a warehouse), or a structure (e.g., a file). For example,

a person or a car (Figure 12.2) can be viewed as a data object in the sense that either can be defined in terms of a set of attributes. The data object description incorporates the data object and all of its attributes.

2. Data Attributes

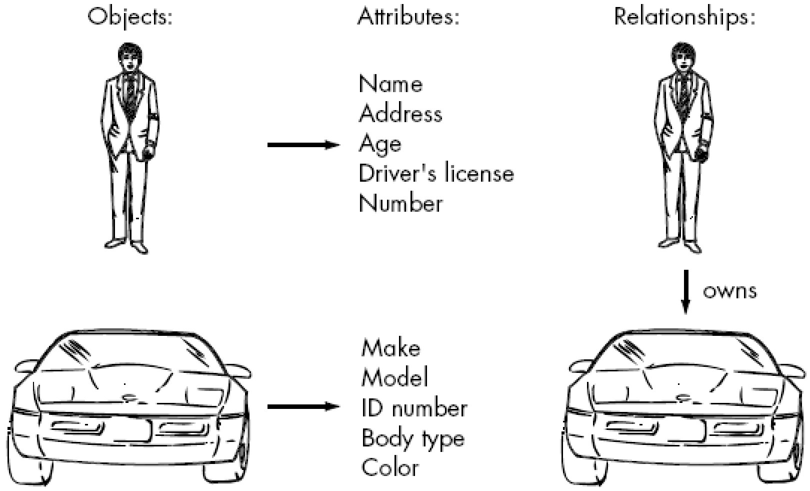
Attributes define the properties of a data object and take on one of three

different characteristics. They can be used to

(1) name an instance of the data object,

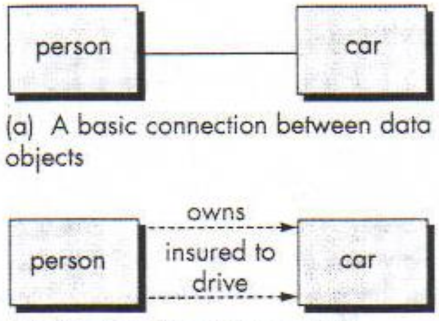
(2) describe the instance, or

(3) make reference to another instance in another table.



3. Relationships

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



(a) A basic connection between data objects

(b) Relationships between data objects

4. Cardinality and Modality

The elements of data modeling—data objects, attributes, and relationships— provide the basis for understanding the information domain of a problem. However, additional information related to these basic elements must also be understood.

We have defined a set of objects and represented the object/relationship pairs that bind them. But a simple pair that states: **object X** relates to **object Y** does not provide

enough information for software engineering purposes. We must understand how many occurrences of **object X** are related to how many occurrences of **object Y.** This leads to a data modeling concept called cardinality.

Cardinality is the specification of the number of occurrences of one [object] that can be related to the number of occurrences of another [object].

Cardinality defines "the maximum number of objects that can participate in a relationship"

Modality

The modality of a relationship is 0 if there is no explicit need for the relationship to occur or the relationship is optional. The modality is 1 if an occurrence of the relationship is mandatory.

Flow-Oriented Modeling

The DFD takes an input-process-output view of a system. That is, data objects flow into the software, are transformed by processing elements, and resultant data objects flow out of the software. Data objects are represented by labeled arrows and the transformations are represented by circles (also called bubbles). The DFD is presented in hierarchical fashion. That is, the first data flow model sometimes called a level 0 DFD or context diagram represent the system as a whole.

1. Creating a data flow model

The data flow diagram enables the software engineer to develop models of the information domain and functional domain at the same time. As the DFD is refined into greater levels of detail, the analyst performs an implicit functional decomposition of the system.

Guidelines

1. The level 0 data flow diagram should depict the software/system as a single bubble

2. Primary input and output should be carefully noted

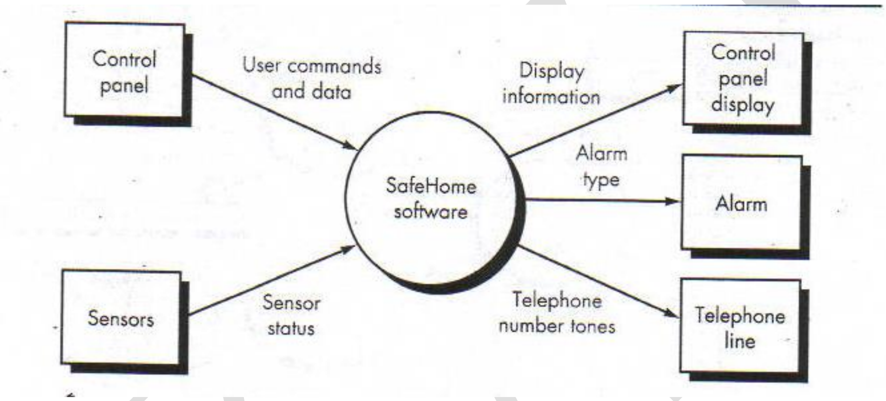
3. Refinement should begin by isolating candidate processes, data objects, and data stores to be represented at the next level

4. All arrows and bubbles should be labeled with meaningful names

5. Information flow continuity must be maintained from level to level

6. One bubble at a time should be refined.

Context level DFD for the safe home security function



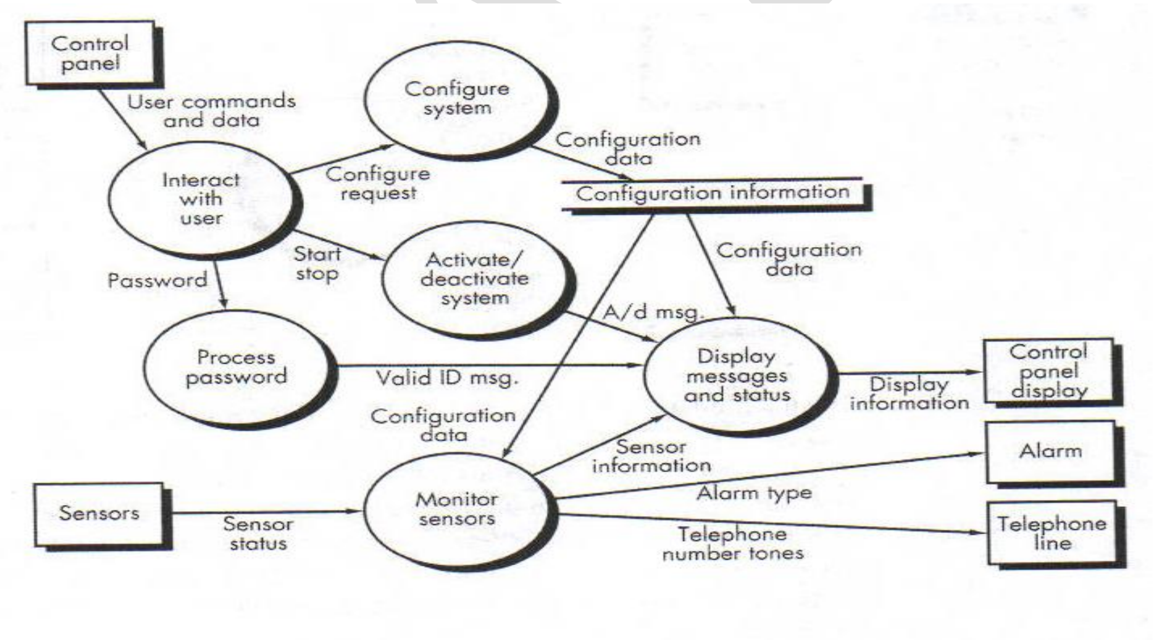
The safe home security function enables the homeowner to configure the security system. When it is installed, monitors all sensors connected to the security system, and interacts with the homeowner through the internet, a PC, or a control panel

During installation, the safe home PC is used to program and configure the system. Each sensor is assigned a number and type, a master password is programmed for arming and disarming the system, and telephone number(s) are input for dialing when a sensor event occurs.

When a sensor event is recognized, the software involves an audible alarm attached to the system. After a delay time that is specified by the homeowner during system configuration activities, the software dials a telephone number of a monitoring service, provides information about the location, reporting the nature of the event that has been detected. The telephone number will be redialed every 20 seconds until a telephone connection is obtained

The level 0 DFD is now expanded into a level 1 data flow model

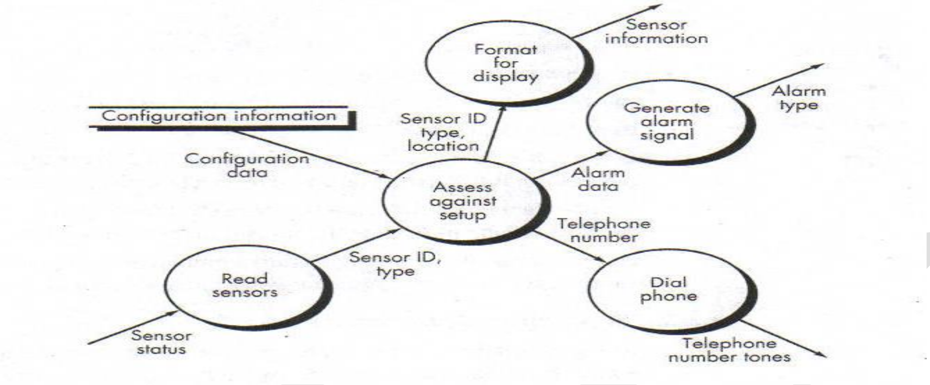
Level 1 DFD for the safe home security function



The homeowner receives security information via a control panel, the PC, or a browser, collectively called an interface. The interface displays prompting messages and system status information on the control panel, the PC, or the browser window

The process represented at DFD level 1 can be further refined into lower levels. For example, the process monitor sensors can be refined into a level 2 DFD.

Level 2 DFD that refines the monitor sensors process



The refinement of DFDs continues until each bubble performs a single function. That is, until the process represented by the bubble performs a function that would be easily implemented as a program component

2. Creating a control flow model

For many types of applications, the data model and the data flow diagram are all that is necessary to obtain meaningful insight into software requirements. As we have already noted, however, a large class of applications are driven by events rather than data, produce control information rather than reports or displays, and process information with heavy concern for time and performance. Such applications require the use of control flow modeling in addition to data flow modeling

To select potential candidate events, the following guidelines are suggested:

- List all sensors that are read by the software
- List all interrupt conditions
- List all switches that are actuated by an operator
- List all data conditions
- Describe the behavior of a system by identifying its states, identify how each state is reached and define the transitions between states.
- Focus on possible omissions

Software Requirement Specification (SRS)

- The requirements are specified in specific format known as SRS.
- This document is created before starting the development work.
- The software requirement specification is an official document.
- It shows the detail about the performance of expected system.
- SRS indicates to a developer and a customer what is implemented in the software.
- SRS is useful if the software system is developed by the outside contractor.
- SRS must include an interface, functional capabilities, quality, reliability, privacy etc.

Characteristics of SRS

- The SRS should be complete and consistence.
- The modification like logical and hierarchical must be allowed in SRS.
- The requirement should be easy to implement.
- Each requirement should be uniquely identified.
- The statement in SRS must be unambiguous means it should have only one meaning.
- All the requirement must be valid for the specified project.

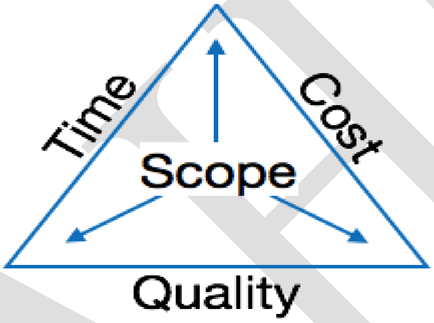
Software Project Management

Software Project

A Software Project is the complete procedure of software development from requirement gathering to testing and maintenance, carried out according to the execution methodologies, in a specified period of time to achieve intended software product.

Need of software project management

Software is said to be an intangible product. Software development is a kind of all new stream in world business and there's very little experience in building software products. Most software products are tailor made to fit client's requirements. The most important is that the underlying technology changes and advances so frequently and rapidly that experience of one product may not be applied to the other one. All such business and environmental constraints bring risk in software development hence it is essential to manage software projects efficiently.



The image above shows triple constraints for software projects. It is an essential part of software organization to deliver quality product, keeping the cost within client's budget constrain and deliver the project as per scheduled. There are several factors, both internal and external, which may impact this triple constrain triangle. Any of three factor can severely impact the other two.

Therefore, software project management is essential to incorporate user requirements along with budget and time constraints.

Software Project Manager

A software project manager is a person who undertakes the responsibility of executing the software project. Software project manager is thoroughly aware of all the phases of SDLC that the software would go through. Project manager may never directly involve in producing the end product but he controls and manages the activities involved in production.

A project manager closely monitors the development process, prepares and executes various plans, arranges necessary and adequate resources, maintains communication among all team members in order to address issues of cost, budget, resources, time, quality and customer satisfaction.

Let us see few responsibilities that a project manager shoulders -

Managing People

- Act as project leader
- Liaison with stakeholders
- Managing human resources
- Setting up reporting hierarchy etc.

Managing Project

- Defining and setting up project scope
- Managing project management activities
- Monitoring progress and performance
- Risk analysis at every phase
- Take necessary step to avoid or come out of problems
- Act as project spokesperson

Software Management Activities

Software project management comprises of a number of activities, which contains planning of project, deciding scope of software product, estimation of cost in various terms, scheduling of tasks and events, and resource management. Project management activities may include:

- **Project Planning**
- **Scope Management**
- **Project Estimation**

Project Planning

Software project planning is task, which is performed before the production of software actually starts. It is there for the software production but involves no concrete activity that has any direction connection with software production; rather it is a set of multiple processes, which facilitates software production. Project planning may include the following:

Scope Management

It defines the scope of project; this includes all the activities, process need to be done in order to make a deliverable software product. Scope management is essential because it creates boundaries of the project by clearly defining what would be done in the project and what would not be done. This makes project to contain limited and quantifiable tasks, which can easily be documented and in turn avoids cost and time overrun.

During Project Scope management, it is necessary to -

- Define the scope
- Decide its verification and control
- Divide the project into various smaller parts for ease of management.
- Verify the scope
- Control the scope by incorporating changes to the scope

Estimation in project planning process

Project Estimation

For an effective management accurate estimation of various measures is a must. With correct estimation managers can manage and control the project more efficiently and effectively.

Project estimation may involve the following:

- **Software size estimation**

  Software size may be estimated either in terms of KLOC (Kilo Line of Code) or by calculating number of function points in the software. Lines of code depend upon coding practices and Function points vary according to the user or software requirement.

  Effort estimation

  The managers estimate efforts in terms of personnel requirement and man-hour required to produce the software. For effort estimation software size should be known. This can either be derived by managers' experience, organization's historical data or software size can be converted into efforts by using some standard formulae.

- **Time estimation**

Once size and efforts are estimated, the time required to produce the software can be estimated. Efforts required is segregated into sub categories as per the requirement specifications and interdependency of various components of software. Software tasks are divided into smaller tasks, activities or events by Work Breakthrough Structure (WBS). The tasks are scheduled on day-to-day basis or in calendar months.

The sum of time required to complete all tasks in hours or days is the total time invested to complete the project.

- **Cost estimation**

  This might be considered as the most difficult of all because it depends on more elements than any of the previous ones. For estimating project cost, it is required to consider -

  - Size of software
  - Software quality
  - Hardware
  - Additional software or tools, licenses etc.
  - Skilled personnel with task-specific skills
  - Travel involved
  - Communication
  - Training and support

➢ **Project Estimation Techniques**

We discussed various parameters involving project estimation such as size, effort, time and cost.Project manager can estimate the listed factors using two broadly recognized techniques –

➢ **Decomposition Technique**

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.

❖ **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;

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

In the context of project planning, size refers to a quantifiable outcome of the software project. If a direct approach is taken, size can be measured in lines of code (LOC). If an indirect approach is chosen, size is represented as function points (FP). Size can be estimated by considering the type of project and its application domain, the functionality delivered (i.e., the number of function points), the number of components to be delivered, the degree to which a set of existing components must be modified for the new system.

- **Problem-Based Estimation**

Lines of code and function points were described as measures from which productivity metrics can be computed. LOC and FP data are used in two ways during software project estimation:

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

(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. LOC or FP (the estimation variable) is then estimated for each function. Baseline productivity metrics (e.g., LOC/pm or FP/pm)6 are then applied to the appropriate estimation variable, and cost or effort for the function is derived. Function estimates are combined to produce an overall estimate for the entire project.

- **Process-Based Estimation**

The most common technique for estimating a project is to base the estimate on the process that will be used. That is, the process is decomposed into a relatively small set of activities, actions, and tasks and the effort required to accomplish each is estimated. Like the problem-based techniques, process-based estimation begins with a delineation of software functions obtained from the project scope. A series of framework activities must be performed for each function.

- **Empirical Estimation Technique**

This technique uses empirically derived formulae to make estimation.These formulae are based on LOC or FPs.

- **Putnam Model**

  This model is made by Lawrence H. Putnam, which is based on Norden's frequency distribution (Rayleigh curve). Putnam model maps time and efforts required with software size.

- **COCOMO**
- COCOMO stands for COnstructive COst MOdel, developed by Barry W. Boehm. It divides the software product into three categories of software: organic, semi-detached and embedded.

Project Scheduling

Project Scheduling in a project refers to roadmap of all activities to be done with specified order and within time slot allotted to each activity. Project managers tend to define various tasks, and project milestones and them arrange them keeping various factors in mind. They look for tasks lie in critical path in the schedule, which are necessary to complete in specific manner (because of task interdependency) and strictly within the time allocated. Arrangement of tasks which lies out of critical path are less likely to impact over all schedule of the project.

Basic Principles

Compartmentalization.The project must be compartmentalized into a number of manageable activities and tasks. To accomplish compartmentalization, both the product and the process are decomposed.

Interdependency.The interdependency of each compartmentalized activity or task must be determined. Some tasks must occur in sequence, while others can occur in parallel. Some activities cannot commence until the work product produced by another is available. Other activities can occur independently.

Time allocation.Each task to be scheduled must be allocated some number of work units (e.g., person-days of effort). In addition, each task must be assigned a start date and a completion date that are a function of the interdependencies and whether work will be conducted on a full-time or part-time basis.

Effort validation*:* Every project has a defined number of people on the software team. As time allocation occurs, no more than the allocated number of people has been scheduled at any given time.

Defined responsibilities: Every task that is scheduled should be assigned to a specific team member.

Defined outcomes*:* Every task that is scheduled should have a defined outcome. For software projects, the outcome is normally a work product (e.g., the design of a component) or a part of a work product. Work products are often combined in deliverables.

Defined milestones: Every task or group of tasks should be associated with a project milestone. A milestone is accomplished when one or more work products has been reviewed for quality and has been approved.

Possible Questions

Part – B (2 Mark)

1. What is requirement engineering process?
2. List the analysis rules of thumb.
3. Draw the context level DFD of Safe Home Alarm System.
4. What are the four major approaches used in analysis modeling
5. Differentiate between cardinality and modality

Part – C (6 Mark)

1. Illustrate the Guidelines for drawing a DFD and explain it with an example.
2. Describe Sequence diagram with an example.
3. Compare and contrast Process Specification and Control Specification.
4. Illustrate the identification of Events with Use-Case while creating a Behavioral Model.
5. Write Short notes on Attributes and Relationships.
6. Give a detailed note on active state and passive state.
7. Discuss the Control Specification in Flow Oriented Modeling.
8. Narrate the steps involved in creating a behavioral model.
9. Describe in detail about analysis modeling concepts.
10. Elucidate Requirement Analysis process in analysis model.
11. Explain State diagram with an example.
12. Compare Cardinality and Modality.
13. Describe the Analysis Modeling Approaches and explain the Rules of Thumb.
14. Elucidate Creation of Flow Oriented Modeling in software engineering.

KARPAGAM ACADEMY OF HIGHER EDUCATION

DEPARTMENT OF COMPUTER SCIENCE

II BSc CS BATCH(2016 - 2019)

PART-A OBJECTIVE TYPE/ MULTIPLE CHOICE QUESTIONS

UNIT-2

S.NO	Questions	Opt1	Opt2	Opt3	Opt4	Answer
1	_____________ is a process of discovery, refinement, modeling, and specification	software engineering	software requirement engineering	software analysis	software design	software engineering
2	____________ is the systematic use of proven principles, techniques, languages, and tools.	software engineering	software analysis	software design	requirements engineering	requirements engineering
3	Requirement engineering is conducted in a _______________.	sporadic way	random way	haphazard way	systematic use of proven approaches	systematic use of proven approaches
4	Software requirements analysis work products must be reviewed for _________.	modeling	completeness	information processing	functional requirement	completeness
5	. __________ bridges the gap between system level requirement engineering and software design.	system engineering	modeling	requirements analysis	software engineering	software engineering
6	Software requirements analysis is divided into __________ areas of effort.	2	3	4	5	4
7	Throughout evaluation and solution synthesis, the analyst’s primary focus is on ___________.	when	where	what	how	what
8	Software applications can be collectively called as ______________.	data gathering	information gathering	data processing	information processing	data processing
9	__________ represents the individual data and control objects that constitute some larger collection of information transformed by the software.	information content	data content	data model	information model	information content
10	__________ represents the manner in which data and control change as each moves through a system.	information content	information flow	information structure	data structure	information flow
11	____________ represents the internal organization of various data and control items.	information content	information flow	information structure	data structure	information structure
12	Entity is a __________.	data	information	model	physical thing	physical thing
13	The first operational analysis principle requires an examination of the information domain and the creation of a .	data model	information model	data structure	information structure	data model
14	To transform software into information, the system performs _____________.	input	processing	output	input, processing and output	input, processing and output
15	. To transform software into information, the system must perform _________ generic functions.	2	3	4	5	3
16	There are ___________ types of models.	5	4	3	2	2
17	The horizontal partitioning of SafeHome function has _____________ major functions on the first level of hierarchy.	2	3	4	5	3
18	The vertical partitioning of SafeHome function has _____________ major functions on the first level of hierarchy.	2	3	4	5	3
19	A model of the software to be built is called ______________.	data model	prototype	information model	software model	prototype
20	The ___________ of software requirements presents the real world manifestation of processing functions and information structures	implementation view	essential view	partitioning view	evolutionary view	implementation view
21	. The essential view of the SafeHome function _______________ does not concern itself with the physical form of the data that is used.	identify event	read sensor status	activate sensor	deactivate sensor	read sensor status

22	A prototype is the ____________.	data model	information model	software model	evolution model	software model
23	Data objects are represented by ________	labeled arrows	bubbles	entity	label	labeled arrows
24	Transformations are represented by ____	labeled arrows	bubbles	entity	label	bubbles
25	__________ enables the software engineer to generate executable code quickly, they are ideal for rapid prototyping.	2 GT	3 GT	4 GT	5 GT	4 GT
26	The ___________ provides a detailed description of the problem that the software must solve.	information description	software scope	function description	software description	information description
27	______________ is probably the most important and, ironically, the most often neglected section of software requirements specification.	behavioural description	processing narrative	overall structure	validation criteria	validation criteria
28	The software requirements specification includes _____________.	bibliography	appendix	Bibliography and appendix	review	Bibliography and appendix
29	The __________ section of the specification examines the operation of the software as a consequence of external events and internally generated control characteristics.	behavioural description	representation format	specification principles	prototyping environment	behavioural description
30	The software requirements specification is developed as a consequence of __________.	review	analysis	prototyping	functional description	analysis
31	The preliminary user's manual presents the software as a _____________.	white box	black box	machine interface	prototype	black box
32	_____________ is the first technical step in the software process.	requirements analysis	requirements specification	information description	information domain	requirements analysis
33	The close ended approach of the prototyping paradigm is called ________.	evolutionary prototyping	simply prototyping	open ended prototyping	throwaway prototyping	throwaway prototyping
34	The information domain contains ___________ different views of the data and control as each is processed by a computer program.	2	3	4	5	3
35	The content of ___________ is defined by the attributes that are needed to create it.	system status	functional model	paycheck	behavioural model	paycheck
36	Building data, functional and behavioural models provide the software engineer with ________ different views	5	4	3	2	3
37	The description of each function required to solve the problem is presented in the _____________.	functional description	behavioural description	data description	program description	functional description
38	Software requirements analysis work products must be reviewed for ___________.	clarity	completeness	consistency	all of the above	all of the above
39	The overall role of software in a larger system is identified during the _________.	system engineering	software planning	software estimation	documentation	system engineering
40	The analyst finds that problems with the current manual system include _________.	inability to obtain the status of a	two-or-three day turn around to	multiple reorders to the same	all of the above	all of the above
41	The expansion of FAST is _______________.	Facilitated Application Specification	Fast Application Specification	Facilitated Application Software	Facilitated Application System	Facilitated Application Specification
42	The following come under the lists of constraints.	cost	size	business rules	all of the above	all of the above
43	All analysis methods are related by a set of operational _____________.	system	software	principles	analysis	principles
44	The functions that the software is to perform must be ______________.	defined	described	discussed	listed	defined
45	The first step in establishing traceability back to the customer is __________.	use multiple views of requirement	rank requirement	record the origin of and the reason for	work to eliminate ambiguity	record the origin of and the reason for
46	___________ are used so that the characteristics of function and behaviour can be communicated in a compact fashion.	softwares	models	programs	none of the above	models

47	The perception of the quality software is often based on the perception of the “friendliness” of the ____________, prototyping are highly recommended.	system	software	interface	prototype	interface
48	All software applications collectively called __________.	packages	programs	software	data processing	data processing
49	The information domain contains __________ different views of data and control as each is processed by a computer program.	2	3	4	5	3
50	The fourth operational analysis principle suggests that the informational, functional and behavioural domains of software can be ____________ .	decomposed	partitioned	listed	described	partitioned
51	The _____________ aids the analyst in understanding the information, function and behaviour of a system, thereby making the requirements analysis task easier and more	prototype	software	model	interface	model
52	The __________ becomes the focal point for review and, therefore the key to a determination of completeness, consistency, and accuracy of the specifications.	prototype	interface	software	model	model
53	The __________ becomes the foundation for design, providing the designer with an essential representation of software that can be mapped in to an implementation context.	prototype	model	interface	software	model
54	the __________ is one method for representing the behavior of a system by depicting its state and evevts	state diagram	use case diagram	ER diagram	DFD	state diagram
55	When an sensor event is recognized, the ___________ invokes an audible alarm attached to the system.	model	software	delay	prototype	software
56	A _________ is always a model – an abstraction of some real situation that is normally quite complex.	software	prototype	specification	function	specification
57	The _____________ presents the software as a black box.	preliminary user’s manual	prototype	system	software	preliminary user’s manual
58	___________ may be accompanied by an executable prototype, a paper prototype or a preliminary user’s manual.	system	software requirements specification	software	user manual	software requirements specification
59	What are the types of requirements ?	Availability	Reliability	Usability	All of the mentioned	All of the mentioned
60	Select the developer specific requirement ?	Potability	Maintainability	Availability	Both Potability and Maintainability	Both Potability and Maintainability

UNIT-III

SYLLABUS

Risk Management: Software Risks, Risk Identification Risk Projection and Risk Refinement, RMMM plan, **Quality Management-** Quality Concepts, Software Quality Assurance, Software Reviews, Metrics for Process and Projects.

Software Risks

Although there has been considerable debate about the proper definition for software risk, there is general agreement that risk always involves two characteristics: uncertainty—the risk may or may not happen; that is, there are no 100 percent probable risks1—and loss—if the risk becomes a reality, unwanted consequences or losses will occur. When risks are analyzed, it is important to quantify the level of uncertainty and the degree of loss associated with each risk. To accomplish this, different categories of risks are considered.

Project risks threaten the project plan. That is, if project risks become real, it is likely that the project schedule will slip and that costs will increase. Project risks identify potential budgetary, schedule, personnel (staffing and organization), resource, stakeholder, and requirements problems and their impact on a software project. Project complexity, size, and the degree of structural uncertainty were also defined as project (and estimation) risk factors.

Technical risks threaten the quality and timeliness of the software to be produced. If a technical risk becomes a reality, implementation may become difficult or impossible. Technical risks identify potential design, implementation, interface, verification, and maintenance problems. In addition, specification ambiguity, technical uncertainty, technical obsolescence, and "leading-edge" technology are also risk factors. Technical risks occur because the problem is harder to solve than you thought it would be.

Business risks threaten the viability of the software to be built and often jeopardize the project or the product. Candidates for the top five business risks are (1) building an excellent product or system that no one really wants (market risk), (2) building a product that no longer fits into the overall business strategy for the company (strategic risk), (3) building a product that the sales force doesn’t understand how to sell (sales risk), (4) losing the support of senior management due to a change in focus or a change in people (management risk), and (5) losing budgetary or personnel commitment (budget risks).

Known risks are those that can be uncovered after careful evaluation of the project plan, the business and technical environment in which the project is being developed, and other reliable information sources (e.g., unrealistic delivery date, lack of documented requirements or software scope, poor development environment).

Predictable risks are extrapolated from past project experience (e.g., staff turnover, poor communication with the customer, dilution of staff effort as ongoing maintenance requests are serviced). Unpredictable risks are the joker in the deck. They can and do occur, but they are extremely difficult to identify in advance.

Risk Identification

Risk identification is a systematic attempt to specify threats to the project plan (estimates, schedule, resource loading, etc.). By identifying known and predictable risks, the project manager takes a first step toward avoiding them when possible and controlling them when necessary.

There are two distinct types of risks for each of the categories

Generic risks are a potential threat to every software project.

Product-specific risks can be identified only by those with a clear understanding of the technology, the people, and the environment that is specific to the software that is to be built.

One method for identifying risks is to create a risk item checklist. The checklist can be used for risk identification and focuses on some subset of known and predictable risks in the following generic subcategories:

Product size—Risks associated with the overall size of the software to be built or modified.

Business impact—Risks associated with constraints imposed by management or the marketplace.

Stakeholder characteristics—Risks associated with the sophistication of the stakeholders and the developer’s ability to communicate with stakeholders in a timely manner.

Process definition—Risks associated with the degree to which the software process has been defined and is followed by the development organization.

Development environment—Risks associated with the availability and quality of the tools to be used to build the product.

Technology to be built—Risks associated with the complexity of the system to be built and the "newness" of the technology that is packaged by the system.

Staff size and experience—Risks associated with the overall technical and project experience of the software engineers who will do the work.

Assessing Overall Project Risk

The following questions have been derived from risk data obtained by surveying experienced Software project managers in different parts of the world

The questions are ordered by their relative importance to the success of a project.

1. Have top software and customer managers formally committed to support the project?

2. Are end users enthusiastically committed to the project and the system/product to be built?

3. Are requirements fully understood by the software engineering team and its customers?

4. Have customers been involved fully in the definition of requirements?

5. Do end users have realistic expectations?

6. Is the project scope stable?

7. Does the software engineering team have the right mix of skills?

8. Are project requirements stable?

9. Does the project team have experience with the technology to be implemented?

10. Is the number of people on the project team adequate to do the job?

11. Do all customer/user constituencies agree on the importance of the project and on the requirements for the system/product to be built?

Risk Components and Drivers

The risk components are defined in the following manner:

- Performance risk—The degree of uncertainty that the product will meet its requirements and be fit for its intended use.
- Cost risk—The degree of uncertainty that the project budget will be maintained.
- Support risk—The degree of uncertainty that the resultant software will be easy to correct,

adapt, and enhance.

• Schedule risk—The degree of uncertainty that the project schedule will be maintained and that the product will be delivered on time.

Risk Projection

Risk projection, also called risk estimation, attempts to rate each risks in two ways—(1) the likelihood or probability that the risk is real and will occur and (2) the consequences of the problems associated with the risk, should it occur. You work along with other managers and technical staff to perform four risk projection steps:

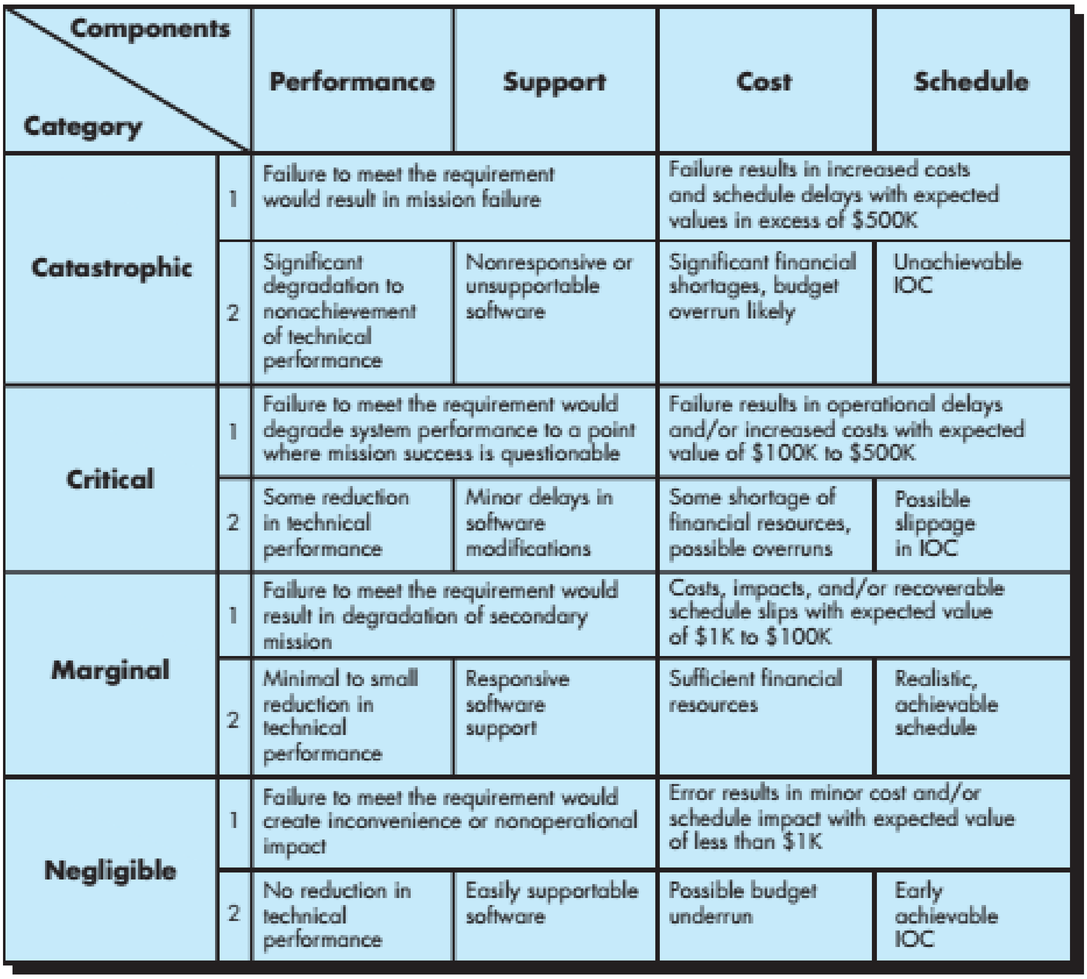
1. Establish a scale that reflects the perceived likelihood of a risk.

2. Delineate the consequences of the risk.

3. Estimate the impact of the risk on the project and the product.

4. Assess the overall accuracy of the risk projection so that there will be no misunderstandings.

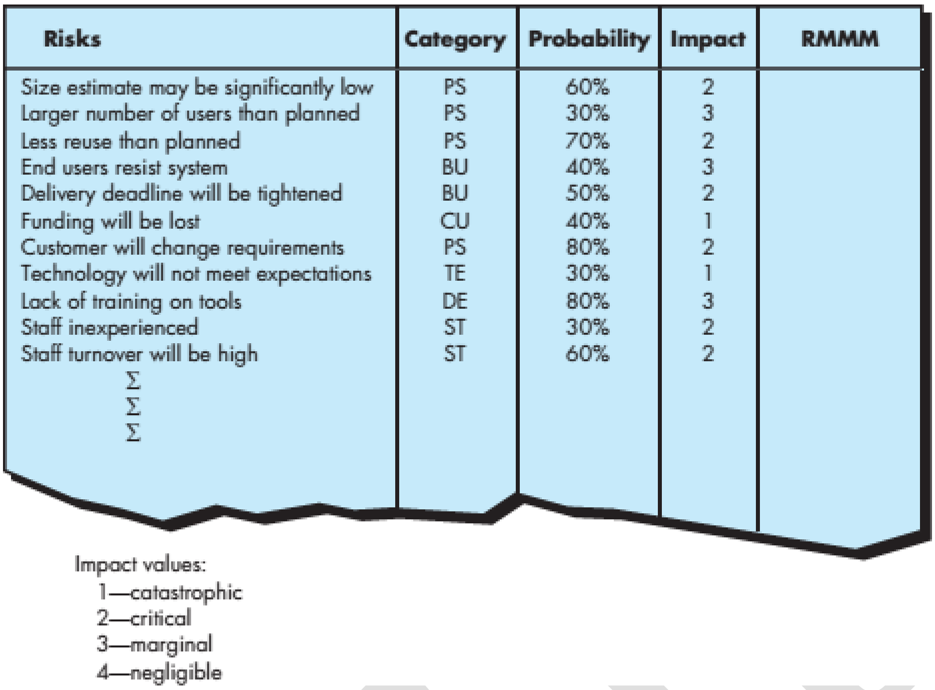
The intent of these steps is to consider risks in a manner that leads to prioritization. No software team has the resources to address every possible risk with the same degree of rigor. By prioritizing risks, you can allocate resources where they will have the most impact.



Components / Category		Performance	Support	Cost	Schedule
Catastrophic	1	Failure to meet the requirement would result in mission failure		Failure results in increased costs and schedule delays with expected values in excess of $500K	
	2	Significant degradation to nonachievement of technical performance	Nonresponsive or unsupportable software	Significant financial shortages, budget overrun likely	Unachievable IOC
Critical	1	Failure to meet the requirement would degrade system performance to a point where mission success is questionable		Failure results in operational delays and/or increased costs with expected value of $100K to $500K	
	2	Some reduction in technical performance	Minor delays in software modifications	Some shortage of financial resources, possible overruns	Possible slippage in IOC
Marginal	1	Failure to meet the requirement would result in degradation of secondary mission		Costs, impacts, and/or recoverable schedule slips with expected value of $1K to $100K	
	2	Minimal to small reduction in technical performance	Responsive software support	Sufficient financial resources	Realistic, achievable schedule
Negligible	1	Failure to meet the requirement would create inconvenience or nonoperational impact		Error results in minor cost and/or schedule impact with expected value of less than $1K	
	2	No reduction in technical performance	Easily supportable software	Possible budget underrun	Early achievable IOC

Developing a Risk Table

A risk table provides you with a simple technique for risk projection. A sample risk table is illustrated in below figure.



Risks	Category	Probability	Impact	RMMM
Size estimate may be significantly low	PS	60%	2	
Larger number of users than planned	PS	30%	3	
Less reuse than planned	PS	70%	2	
End users resist system	BU	40%	3	
Delivery deadline will be tightened	BU	50%	2	
Funding will be lost	CU	40%	1	
Customer will change requirements	PS	80%	2	
Technology will not meet expectations	TE	30%	1	
Lack of training on tools	DE	80%	3	
Staff inexperienced	ST	30%	2	
Staff turnover will be high	ST	60%	2	
Σ				
Σ				
Σ				

Impact values:
1—catastrophic
2—critical
3—marginal
4—negligible

- Begin by listing all risks (no matter how remote) in the first column of the table.
- Each risk is categorized in the second column (e.g., PS implies a project size risk, BU implies a business risk).
- The probability of occurrence of each risk is entered in the next column of the table.
- The probability value for each risk can be estimated by team members individually.
- Each risk component is assessed and an impact category is determined.
- The categories for each of the four risk components—performance, support, cost, and schedule—are averaged to determine an overall impact value.
- Once the first four columns of the risk table have been completed, the table is sorted by probability and by impact.
- High-probability, high-impact risks percolate to the top of the table, and low-probability risks drop to the bottom.
- This accomplishes first-order risk prioritization.

Assessing Risk Impact

Three factors affect the consequences that are likely if a risk does occur: its nature, its scope, and its timing.

- The nature of the risk indicates the problems that are likely if it occurs.
- The scope of a risk combines the severity with its overall distribution

Finally, the timing of a risk considers when and for how long the impact will be felt.

Risk Refinement

During early stages of project planning, a risk may be stated quite generally. As time passes and more is learned about the project and the risk, it may be possible to refine the risk into a set of more detailed risks, each somewhat easier to mitigate, monitor, and manage.
This general condition can be refined in the following manner:

Sub condition 1. Certain reusable components were developed by a third party with no knowledge of internal design standards.
Sub condition 2. The design standard for component interfaces has not been solidified and may
not conform to certain existing reusable components.
Sub condition 3. Certain reusable components have been implemented in a language that is not
supported on the target environment.

The consequences associated with these refined sub conditions remain the same (i.e., 30 percent of software components must be custom engineered), but the refinement helps to isolate the underlying risks and might lead to easier analysis and response.

The RMMM Plan

A risk management strategy can be included in the software project plan, or the risk management steps can be organized into a separate risk mitigation, monitoring and management plan. The RMMM plan documents all work performed as part of risk analysis and are used by the project manager as part of the overall project plan.
Some software teams do not develop a formal RMMM document. Rather, each risk is documented individually using a risk information sheet (RIS). In most cases, the RIS is maintained using a database system so that creation and information entry, priority ordering, searches, and other analysis may be accomplished easily. Once RMMM has been documented and the project has begun, risk mitigation and monitoring steps commence. As we have already discussed, risk mitigation is a problem avoidance activity. Risk monitoring is a project tracking activity with three primary objectives: (1) to assess whether predicted risks do, in fact, occur; (2) to ensure that risk aversion steps defined for the risk are being properly applied; and (3) to

collect information that can be used for future risk analysis. In many cases, the problems that occur during a project can be traced to more than one risk. Another job of risk monitoring is to attempt to allocate origin

Quality Management

Quality Concepts

The drumbeat for improved software quality began in earnest as software became increasingly integrated in every facet of our lives. By the 1990s, major corporations recognized that billions of dollars each year were being wasted on software that didn't deliver the features and functionality that were promised. Worse, both government and industry became increasingly concerned that a major software fault might cripple important infrastructure, costing tens of billions more. By the turn of the century, CIO Magazine trumpeted the headline, "Let's Stop Wasting $78 Billion a Year," lamenting the fact that "American businesses spend billions for software that doesn't do what it's supposed to do". InformationWeek echoed the same concern: Despite good intentions, defective code remains the hobgoblin of the software industry, accounting for as much as 45% of computer-system downtime and costing U.S. companies about $100 billion last year in lost productivity and repairs, says the Standish Group, a market research firm. That doesn't include the cost of losing angry customers. Because IT shops write applications that rely on packaged infrastructure software, bad code can wreak havoc on custom apps as well . . . Just how bad is bad software? Definitions vary, but experts say it takes only three or four defects per 1,000 lines of code to make a program perform poorly. Factor in that most programmers inject about one error for every 10 lines of code they write, multiply that by the millions of lines of code in many commercial products, then figure it costs software vendors at least half their development budgets to fix errors while testing. Get the picture? In 2005, Computer World lamented that "bad software plagues nearly every organization that uses computers, causing lost work hours during computer downtime, lost or corrupted data, missed sales opportunities, high IT support and maintenance costs, and low customer satisfaction. A year later, InfoWorld wrote about the "the sorry state of software quality" reporting that the quality problem had not gotten any better. As the emphasis on software quality grew, a survey of 100,000 white-collar professionals indicated that software quality engineers were "the happiest workers in America"! Today, software quality remains an issue, but who is to blame? Customers blame developers, arguing that sloppy practices lead to low-quality software. Developers blame customers (and other stakeholders), arguing that irrational delivery dates and a continuing stream of changes force them to deliver software before it has been fully validated. Who's right? Both—and that's the problem.

Software Quality Assurance

The software engineering approach works toward a single goal: to produce on-time, high-quality software. Yet many readers will be challenged by the question: "What is software quality?"

The problem of quality management is not what people don't know about it. The problem is what they think they do know

Everybody is for it. (Under certain conditions, of course.) Everyone feels they understand it. (Even though they wouldn't want to explain it.) Everyone thinks execution is only a matter of following natural inclinations. (After all, we do get along somehow.) And, of course, most people feel that problems in these areas are caused by other people. (If only they would take the time to do things right.) Indeed, quality is a challenging concept. Some software developers continue to believe that software quality is something you begin to worry about after code has been generated. Nothing could be further from the truth! Software quality assurance (often called quality management) is an umbrella activity that is applied throughout the software process.

Software quality assurance (SQA) encompasses: (1) an SQA process, (2) specific quality assurance and quality control tasks (including technical reviews and a multi-tiered testing strategy), (3) effective software engineering practice (methods and tools), (4) control of all software work products and the changes made to them, (5) a procedure to ensure compliance with software development standards (when applicable), and (6) measurement and reporting mechanisms.

Elements of Software Quality Assurance

Software quality assurance encompasses a broad range of concerns and activities that focus on the management of software quality. These can be summarized in the following manner: **Standards.** The IEEE, ISO, and other standards organizations have produced a broad array of software engineering standards and related documents. Standards may be adopted voluntarily by a software engineering organization or imposed by the customer or other stakeholders. The job of SQA is to ensure that standards that have been adopted are followed and that all work products conform to them.

Reviews and audits. Technical reviews are a quality control activity performed by software engineers for software engineers. Their intent is to uncover errors. Audits are a type of review performed by SQA personnel with the intent of ensuring that quality guidelines are being followed for software engineering work. For example, an audit of the review process might be conducted to ensure that reviews are being performed in a manner that will lead to the highest likelihood of uncovering errors.

Testing. Software testing is a quality control function that has one primary goal—to find errors. The job of SQA is to ensure that testing is properly planned and efficiently conducted so that it has the highest likelihood of achieving its primary goal.

Error/defect collection and analysis. The only way to improve is to measure how you're doing. SQA collects and analyzes error and defect data to WebRef.

Software Quality Assurance

Better understand how errors are introduced and what software engineering activities are best suited to eliminating them.

Change management. Change is one of the most disruptive aspects of any software project. If it is not properly managed, change can lead to confusion, and confusion almost always leads to poor quality. SQA ensures that adequate change management practices have been instituted.

Education. Every software organization wants to improve its software engineering practices. A key contributor to improvement is education of software engineers, their managers, and other stakeholders. The SQA organization takes the lead in software process improvement and is a key proponent and sponsor of educational programs.

Vendor management. Three categories of software are acquired from external software vendors—shrink-wrapped packages (e.g., Microsoft Office), a tailored shell that provides a basic skeletal structure that is custom tailored to the needs of a purchaser, and contracted software that is custom designed and constructed from specifications provided by the customer organization. The job of the SQA organization is to ensure that high-quality software results by suggesting specific quality practices that the vendor should follow (when possible), and incorporating quality mandates as part of any contract with an external vendor. **Security management.** With the increase in cyber crime and new government regulations regarding privacy, every software organization should institute policies that protect data at all levels, establish firewall protection for WebApps, and ensure that software has not been tampered with internally. SQA ensures that appropriate process and technology are used to achieve software security.

Safety. Because software is almost always a pivotal component of human-rated systems (e.g., automotive or aircraft applications), the impact of hidden defects can be catastrophic. SQA may be responsible for assessing the impact of software failure and for initiating those steps required to reduce risk.

Risk management. Although the analysis and mitigation of risk is the concern of software engineers, the SQA organization ensures that risk management activities are properly conducted and that risk-related contingency plans have been established. In addition to each of these concerns and activities, SQA works to ensure that software support activities (e.g., maintenance, help lines, documentation, and manuals) are conducted or produced with quality as a dominant concern.

Software Reviews

Metrics for Process and Projects

Measurement enables us to gain insight into the process and the project by providing a mechanism for objective evaluation. Lord Kelvin once said: When you can measure what you are speaking about and express it in numbers, you know something about it; but when you cannot measure, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind: it may be the beginning of knowledge, but you have scarcely, in your thoughts, advanced to the stage of a science. The software engineering community has taken Lord Kelvin's words to heart. But not without frustration and more than a little controversy! Measurement can be applied to the software process with the intent of improving it on a continuous basis. Measurement can be used throughout a software project to assist in estimation, quality control, productivity assessment, and project control. Finally, measurement can be used by software engineers to help assess the quality of work products and to assist in tactical decision making as a project proceeds. Within the context of the software process and the projects that are conducted using the process, a software team is concerned primarily with productivity and quality metrics—measures of software development "output" as a function of effort and time applied and measures of the "fitness for use" of the work products that are produced. For planning and estimating purposes, our interest is historical. What was software development productivity on past projects? What was the quality of the software that was produced? How can past productivity and quality data are extrapolated to the present? How can it help us plan and estimate more accurately?

The reasons that we measure: (1) to characterize in an effort to gain an understanding "of processes, products, resources, and environments, and to establish baselines for comparisons with future assessments"; (2) to evaluate "to determine status with respect to plans"; (3) to predict by "gaining understandings of relationships among processes and products and building models of these relationships"; and (4) to improve by "identifying roadblocks, root causes, inefficiencies, and other opportunities for improving product quality and process performance."

Measurement is a management tool. If conducted properly, it provides a project manager with insight. And as a result, it assists the project manager and the software team in making decisions that will lead to a successful project.

Metrics in the Process and Project Domains

Process metrics are collected across all projects and over long periods of time. Their intent is to provide a set of process indicators that lead to long-term software process improvement. Project metrics enable a software project manager to (1) assess the status of an ongoing project, (2) track potential risks, (3) uncover problem areas before they go "critical," (4) adjust work flow or tasks, and (5) evaluate the project team's ability to control quality of software work products.

Measures that are collected by a project team and converted into metrics for use during a project can also be transmitted to those with responsibility for software process improvement. For this reason, many of the same metrics are used in both the process and project domains.

Process Metrics and Software Process Improvement

The only rational way to improve any process is to measure specific attributes of the process, develop a set of meaningful metrics based on these attributes, and then use the metrics to provide indicators that will lead to a strategy for improvement (Chapter 37). But before we discuss software metrics and their impact on software process improvement, it is important to note that process is only one of a number of “controllable factors in improving software quality and organizational performance”.

Process sits at the center of a triangle connecting three factors that have a profound influence on software quality and organizational performance. The skill and motivation of people have been shown to be the most influential factors in quality and performance. The complexity of the product can have a substantial impact on quality and team performance. The technology (i.e., the software engineering methods and tools) that populates the process also has an impact. In addition, the process triangle exists within a circle of environmental conditions that include the development environment (e.g., integrated software tools), business conditions (e.g., deadlines, business rules), and customer characteristics (e.g., ease of communication and collaboration). You can only measure the efficacy of a software process indirectly. That is, you derive a set of metrics based on the outcomes that can be derived from the process. Outcomes include measures of errors uncovered before release of the software, defects delivered to and reported by end users, work products delivered (productivity), human effort expended, calendar time used, schedule conformance, and other measures. You can also derive process metrics by measuring the characteristics of specific software engineering tasks. For example, you might measure the effort and time spent performing the umbrella activities and the generic software engineering activities described in

The skill and motivation of the software people doing the work are the most important factors that influence software quality.

“Software metrics let you know when to laugh and when to cry.”

Process Product People Technology

Development
environment
Customer
characteristics
Business
conditions

Managing Software Projects

Grady argues that there are "private and public" uses for different types of process data. Because it is natural that individual software engineers might be sensitive to the use of metrics collected on an individual basis, these data should be private to the individual and serve as an indicator for the individual only. Examples of private metrics include defect rates (by individual), defect rates (by component), and errors found during development. The "private process data" philosophy conforms well with the Personal Software Process approach proposed by Humphrey. Humphrey recognized that software process improvement can and should begin at the individual level. Private process data can serve as an important driver as you work to improve your software engineering approach. Some process metrics are private to the software project team but public to all team members. Examples include defects reported for major software functions (that have been developed by a number of practitioners), errors found during technical reviews and lines of code or function points per component or function.1 The team reviews these data to uncover indicators that can improve team performance. Public metrics generally assimilate information that originally was private to individuals and teams. Project-level defect rates (absolutely not attributed to an individual), effort, calendar times, and related data are collected and evaluated in an attempt to uncover indicators that can improve organizational process performance.

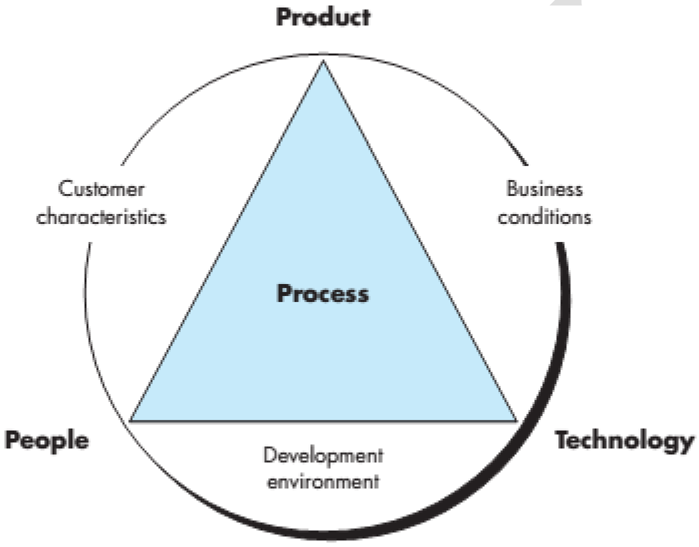
Software process metrics can provide significant benefits as an organization works to improve its overall level of process maturity. However, like all metrics, these can be misused, creating more problems than they solve. Grady suggests a "software metrics etiquette" that is appropriate for both managers and practitioners as they institute a process metrics program:

- Use common sense and organizational sensitivity when interpreting metrics data.
- Provide regular feedback to the individuals and teams who collect measures and metrics.
- Don't use metrics to appraise individuals.
- Work with practitioners and teams to set clear goals and metrics that will be used to achieve them.
- Never use metrics to threaten individuals or teams.
- Metrics data that indicate a problem area should not be considered "negative." These data are merely an indicator for process improvement.

What is the difference between private and public uses for software metrics? What guidelines should be applied when we collect software metrics?

Process and Project Metrics

• Don't obsess on a single metric to the exclusion of other important metrics. As an organization becomes more comfortable with the collection and use of process metrics, the derivation of simple indicators gives way to a more rigorous approach called statistical software process improvement (SSPI). In essence, SSPI uses software failure analysis to collect information about all errors and defects2 encountered as an application, system, or product is developed and used.



Project Metrics

Unlike software process metrics that are used for strategic purposes, software project measures are tactical. That is, project metrics and the indicators derived from them are used by a project manager and a software team to adapt project work flow and technical activities. The first application of project metrics on most software projects occurs during estimation. Metrics collected from past projects are used as a basis from which effort and time estimates are made for current software work. As a project proceeds, measures of effort and calendar time expended are compared to original estimates (and the project schedule). The project manager uses these data to monitor and control progress.

As technical work commences, other project metrics begin to have significance. Production rates represented in terms of models created, review hours, function points, and delivered source lines are measured. In addition, errors uncovered during each software engineering task are tracked. As the software evolves from requirements into design, technical metrics (Chapter 30) are collected to assess design quality and to provide indicators that will influence the approach taken to code generation and testing. The intent of project metrics is twofold. First, these metrics are used to minimize the development schedule by making the adjustments necessary to avoid delays and mitigate potential problems and risks. Second, project metrics are used to assess product quality on an ongoing basis and, when necessary, modify the technical approach to improve quality. As quality improves, defects are minimized, and as the defect count goes down, the amount of rework required during the project is also reduced. This leads to a reduction in overall project cost.

An error is defined as some flaw in a software engineering work product that is uncovered before the software is delivered to the end user. A defect is a flaw that is uncovered after delivery to the end user. It should be noted that others do not make this distinction.

Possible Questions

Part – B (2 Mark)

1. Write about generic risk and product risk.
2. What is risk projection?
3. Define RMMM plan and its use.
4. What are the elements of software quality assurance
5. What are project metrics?

Part – C (6 Mark)

1. Explain in detail about Software risks that are faced by developers.
2. What is the use of software reviews? Explain in detail.
3. Describe in detail how are risks identified.
4. What is risk refinement? Explain in detail the steps to refine a risk if it occurs.
5. Illustrate risk projection mechanism in software engineering
6. What are Formal technical reviews? How are they conducted in software engineering?
7. Describe in detail about RMMM plan
8. Explain in detail about project matrices
9. Enumerate in detail the quality concepts that must be considered in developing a software
10. Explain the software quality assurance standards in detail.

KARPAGAM ACADEMY OF HIGHER EDUCATION

DEPARTMENT OF COMPUTER SCIENCE

II BSc CS BATCH(2016 - 2019)

PART-A OBJECTIVE TYPE/ MULTIPLE CHOICE QUESTIONS

UNIT-3 - SOFTWARE ENGINEERING (16CSU402)

s.no	**Questions**	**Opt1**	**Opt2**	**Opt3**	**Opt4**	**Answer**
1	Which factors affect the probable consequences if a risk occur?	Risk avoidance	Risk monitoring	Risk timing	Contingency planning	Risk timing
2	Staff turnover, poor communication with the customer are risks that are extrapolated from past experience are called _____ .	Business risks	Predictable risks	Project risks	Technical risks	Predictable risks
3	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	Schedule risk
4	Project risk factor is considered in which model?	Spiral model	Waterfall model	Prototyping model	None of the above	Spiral model
5	_________is a systematic attempt to specify threats to the project plan.	Generic risks	Product-specific risks	Risk identification	None of the above	Risk identification
6	_________risks associated with the overall size of the software to be built or modified.	Product size	Business impact risks	Process definition risks	Product Risk	Product size
7	________risks associated with constraints imposed by management or the marketplace.	Business impact risks	Product size	Process definition risks	customer Risk	Business impact risks
8	_________risks associated with the availability and quality of the tools to be used to build the product.	Technical risk	Project risk	Development environment	Risk identification	Development environment
9	Project Risk is directly proportional to ________	project size	product size	Estimated size	size of project	product size
10	The __________ environment supports the project team, the process, and the product.	software engineering	Software plan	software engineer	All the above	software engineering
11	_______ Associated with Staff Size and Experience	Size	Product	Risks	Project risk	Risks
12	The degree of _________ uncertainty that the product will meet its requirements	Cost risk	Support risk	Performance risk	support risk	Performance risk
13	The degree of uncertainty that the ___________ will be maintained.	project budget	Cost risk	Schedule risk	Performance risk	project budget
14	Risk projection, also called _________.	Estimation plan	Estimation Cost	Risk Cost	risk estimation	risk estimation
15	A _____ provides a project manager with a simple technique for risk projection	projection table	cost table	risk table	Project table	risk table
16	The ______implies that only risks that lie above the line will be given further attention.	offline cost	cost line	cut-off line	offline	cut-off line
17	A risk referent level has a single point, called the ________	break point	referent point	risk point	project point	referent point
18	The ________ should monitor the effectiveness of risk mitigation steps.	project manager	engineer	manager	cost manager	project manager
19	________assesses risks that may affect the outcome of the project or the quality of the product.	Time management	Risk management	quality management	cost management	Risk management
20	______ management is one of the key attributes of a successful software project.	Risk	Project	cost	product	Risk
21	The SQA organization ensures that risk management activities are properly conducted and that risk-related have been established.	novices	cut-off line	contingency plans	plans	contingency plans
22	Risk analysis and management are actions that help a _______ to understand and manage uncertainty	project size	project team	team members	software team	software team
23	A risk mitigation,monitoring, and management ______ plan	RMMM	RMMI	RMRM	RMMR	RMMM
24	_________ strategies have been laughingly called the “Indiana Jones school of risk management”	Reactive risk	risk	Reactive	Reactive plan	Reactive risk
25	_________ are identified, their probability and impact are assessed, and they are ranked by importance.	Project risks	Potential risks	project plan	None of the above	Potential risks
26	__________ threaten the project plan	project plan	Project risks	Potential risks	None of the above	Project risks

27	_________ threaten the quality and timeliness of the software to be produced.	Technical risks	Potential risks	Project risks	None of the above	Technical risks
28	_________ threaten the viability of the software to be built and often jeopardize the project or the product.	Business risks	knowledgeable, intermittent users	knowledgeable, frequent users	Testers	Business risks
29	_________ are extrapolated from past project experience	user's model	Predictable risks	design model	system image	Predictable risks
30	A risk that is_____ percent probable is a constraint on the software project.	9	20	90	10	100
31	___________ can be identified only by those with a clear understanding of the technology.	specification	design	Product-specific risks	prototype	Product-specific risks
32	___________ characteristics risks associated with the sophistication of the stakeholders	manager	Stakeholder	engineer	team members	Stakeholder
33	Risk projection, also called _________.	risk estimation	Estimation Cost	Estimated size	estimation	risk estimation
34	In Risk impact, Sixty reusable _________ were planned.	object oriented approach	top down approach	software components	all of the above	software components
35	_________ and contingency planning assumes that mitigation efforts have failed and that the risk has become a reality.	Risk management	management	Risk	None of the above	Risk management
36	Software safety and hazard analysis are _________ assurance activities.	software quality	software	quality	estimation	software quality
37	Some software teams do not develop a formal ________ document.	system response time	variability	RMMM	all of the above	RMMM
38	Once RMMM has been documented and the project has begun, ________and monitoring steps commence.	risk mitigation	mitigation cost	mitigation plan	mitigation	risk mitigation
39	A "plan-do-check-act" cycle that is applied to the quality management elements of a ___________.	integrated help facility	system response time	software project.	all of the above	software project.
40	Everyone involved in the software engineering process is responsible for ________	procedural abstraction	quality	stepwise refinement	decomposition	quality
41	The amount of computing________ and code required by a program to perform its function.	resources	architectural	interface design	all of the above	resources
42	________required to couple one system to another.	Effort	top level management	software engineer	middle level management	Effort
43	The _________ in software development parallels the history of quality in hardware manufacturing.	quality assurance	Effort	data design	code design	quality assurance
44	The ______ serves as the customer's in-house representative	procedural design	component level design	SQA group	code design	SQA group
45	__________are a quality control activity performed by software engineers for software engineers	graphical	Technical reviews	text-based	all of the above	Technical reviews
46	_________ is a quality control function that has one primary goal is to find errors.	Software testing	testing	planning	communication	Software testing
47	_________ is one of the most disruptive aspects of any software project.	processing step	Change	flow of control	start	Change
48	Every software organization wants to improve its ___________ practices	processing step	software engineering	flow of control	start	software engineering
49	Software engineers address ______ by applying solid technical methods and measures.	quality	logical condition	flow of control	start	quality
50	The plan is developed as part of _________ and is reviewed by all stakeholders.	100	project planning	planning	plans	project planning
51	The ______ identifies, documents, and tracks deviations from the process and verifies that corrections have been made.	SQA group	condition	repetition	all of the above	SQA group
52	A ________ should apply limited resources in a way that has the highest likelihood of achieving a high-quality result.	sequence	condition	software team	selection	software team
53	Statistical ______ reflects a growing trend throughout industry to become more quantitative about quality.	sequence	condition	quality assurance	selection	quality assurance
54	Six Sigma is the most widely used strategy for statistical ________ in industry today.	quality assurance	condition	repetition	selection	quality assurance

55	Data complexity provides an indication of the _______ in the internal interface	box diagram	complexity	transition diagram	decision table	complexity
56	The _____ to which two or more classes are similar in terms of their structure, function, behavior, or purpose is indicated by this measure.	Process Design Language	degree	Program Document Language	Program Document Language	degree
57	Depth of the _____ is “the maximum length from the node to the root of the tree”	inheritance tree	tree cost	tree	plans	inheritance tree
58	The total number of operations that are encapsulated within the _________	class	simplicity	ease of editing	maintainability	class
59	_________are the variables defined for a module can be defined as data tokens for the module.	modularity	simplicity	Data tokens	maintainability	Data tokens
60	A variety of_____________ can be computed to determine the complexity of program control flow.	modularity	simplicity	software metrics	maintainability	software metrics

UNIT-IV

SYLLABUS

Design Engineering-Design Concepts, Architectural Design Elements, Software Architecture,Data Design at the Architectural Level and Component Level, Mapping of Data Flow into Software Architecture, Modeling Component Level Design

Design Engineering

4.1 Design within the Context of Software Engineering

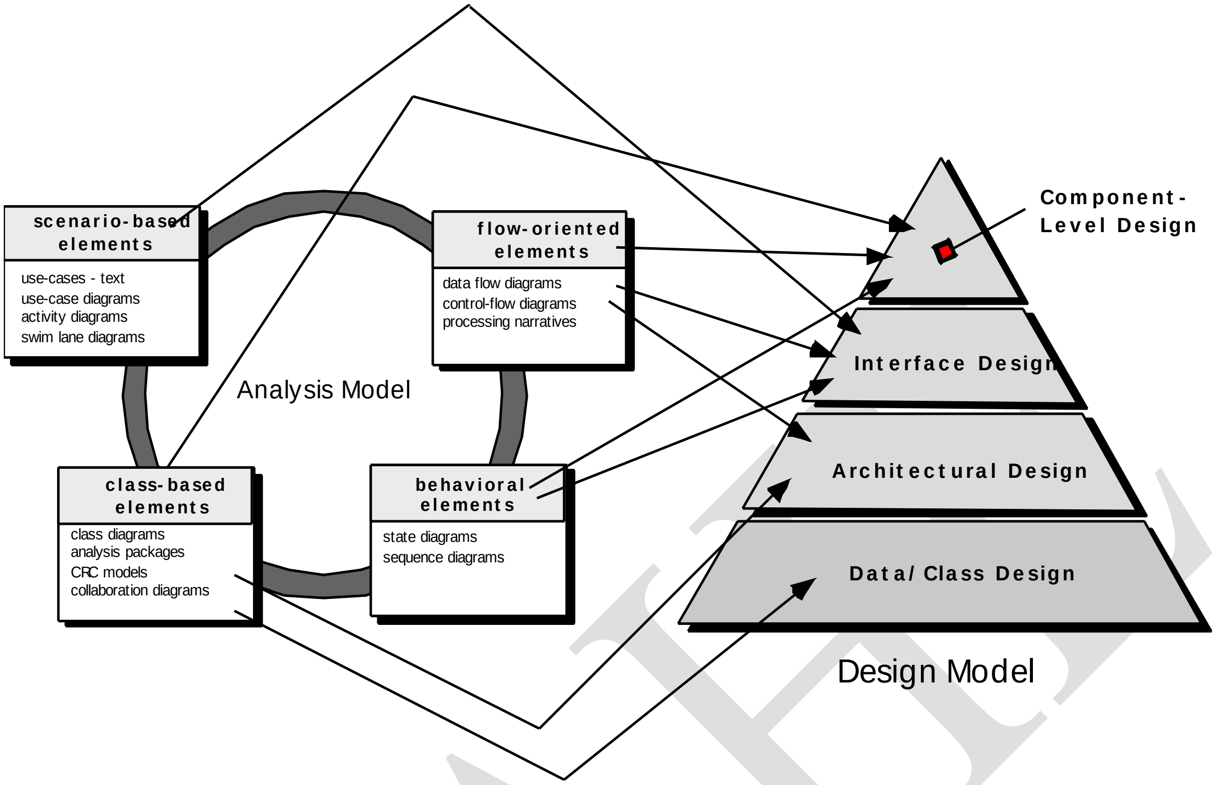
Software design is the last software engineering action within the modeling activity and sets the stage for construction (code generation and testing).

The flow of information during software design is illustrated in Figure below. The analysis model, manifested by scenario-based, class-based, flow-oriented and behavioral elements, feed the design task.

The *architectural design* defines the relationship between more structural elements of the software, the architectural styles and design patterns that can be used to achieve the requirements defined for the system, and the constraints that affect the way in which the architectural design can be implemented.

The *architectural design* can be derived from the System Specs, the analysis model, and interaction of subsystems defined within the analysis model.

The *interface design* describes how the software communicates with systems that interpolate with it, and with humans who use it. An interface implies a flow of information (data, and or control) and a specific type of behavior.



The *component-level design* transforms structural elements of the software architecture into a procedural description of software components

The importance of software design can be stated with a single word – *quality*. Design is the place where quality is fostered in software engineering. Design provides us with representations of software that can be assessed for quality. Design is the only way that we can accurately translate a customer's requirements into a finished software product or system.

4.2 Design Process and Design Quality

Software design is an iterative process through which requirements are translated into a "**blueprint**" for constructing the software.

Initially, the **blueprint** depicts a holistic view of software, i.e. the design is represented at a high-level of abstraction.

Throughout the design process, the quality of the evolving design is assessed with a series of formal technique reviews or design walkthroughs.

Three characteristics serve as a guide for the evaluation of a good design:

- The design must implement all of the explicit requirements contained in the analysis model, and it must accommodate all of the implicit requirements desired by the customer.

- The design must be a readable, understandable guide for those who generate code and for those who test and subsequently support the software.
- The design should provide a complete picture of the software, addressing the data, functional, and behavioral domains from an implementation perspective.

Quality Guidelines

In order to evaluate the quality of a design representation, we must establish technical criteria for good design.

1. A design should exhibit an architecture that:
   (1) Has been created using recognizable architectural styles or patterns,

   (2) Is composed of components that exhibit good design characteristics, and

   (3) Can be implemented in an evolutionary fashion

   a. For smaller systems, design can sometimes be developed linearly.
2. A design should be modular; that is, the software should be logically partitioned into 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 to be implemented and are drawn from recognizable data patterns.
5. A design should lead to components that exhibit independent functional characteristics.
6. A design should lead to interfaces that reduce the complexity of connections between components and with the external environment.
7. A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis.
8. A design should be represented using a notation that effectively communicates its meaning.

Quality Attributes

Hewlett-Packard developed a set of software quality attributes that has been given the acronym FURPS. The FURPS quality attributes represent a target for all software design:

- ✓ *Functionality*: is assessed by evaluating the features set and capabilities of the program, the generality of the functions that are delivered, and the security of the overall system.
- ✓ *Usability*: is assessed by considering human factors, overall aesthetics, consistency, and documentation.
- ✓ *Reliability*: is evaluated by measuring the frequency and severity of failure, the accuracy of output results, the mean-time-to-failure, the ability to recover from failure, and the predictability of the program.

- ✓ *Performance*: is measured by processing speed, response time, resource consumption, throughput, and efficiency.
- ✓ *Supportability*: combines the ability to extend the program extensibility, adaptability, serviceability ➔ maintainability. In addition, testability, compatibility, configurability, etc.

4.3 Design Concepts

This section discusses many significant design concepts (abstraction, refinement, modularity, architecture, patterns, refactoring, functional independence, information hiding, and OO design concepts).

4.3.1 Abstraction

At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment. At lower levels of abstraction, a more detailed description of the solution is provided.

As we move through different levels of abstraction, we work to create procedural and data abstractions. A *procedural abstraction* refers to a sequence of instructions that have a specific and limited function. An example of a procedural abstraction would be the word *open* for a door.

A *data abstraction* is a named collection of data that describes a data object. In the context of the procedural abstraction *open*, we can define a data abstraction called **door**. Like any data object, the data abstraction for **door** would encompass a set of attributes that describe the door (e.g. **door type, swing direction, weight)**.

4.3.2 Architecture

Software architecture alludes to the "overall structure of the software and the ways in which the structure provides conceptual integrity for a system."

In its simplest from, architecture is the structure of organization of program components (modules), the manner in which these components interact, and the structure of data that are used by the components.

Te goal of software design is to derive an architectural rendering of a system. This rendering serves as a framework from which detailed design activities are constructed.

A set of architectural patterns enable a software engineer to reuse design-level concepts.

The architectural design can be represented using one or more of a number of different models.

Structural models represent architecture as an organized collection of program components.

Framework models increase the level of design abstraction by attempting to identify repeatable architectural design frameworks that are encountered in similar types of applications.

Dynamic models address the behavioral aspects of the program architecture, indicating how the structure or system configuration may change as a function of external events.

Process models focus on the design of business or technical process that the system must accommodate.

Functional models can be used to represent the functional hierarchy of a system.

Architectural design will be discussed in Chapter 10.

4.3.3 Patterns

A design pattern "conveys the essence of a proven design solution to a recurring problem within a certain context amidst computing concerns."

A design pattern describes a design structure that solves a particular design problem within a specific context and amid "forces" that may have an impact on the manner in which the pattern is applied and used.

The intent of each design pattern is to provide a description that enables a designer to determine:

1. whether the pattern is applicable to the current work,
2. whether the pattern can be reused, and
3. whether the pattern can serve as a guide for developing a similar, but functionally or structurally different pattern.

4.3.4 Modularity

Software architecture and design patterns embody *modularity*; that is, software is divided into separately named and addressable components, sometimes called modules that are integrated to satisfy problem requirements.

Monolithic software (large program composed of a single module) cannot be easily grasped by a software engineer. The number of control paths, span of reference, number of variables, and overall complexity would make understanding close to impossible.

It is the compartmentalization of data and function. It is easier to solve a complex problem when you break it into manageable pieces. "Divide-and-conquer"

Don't over-modularize. The simplicity of each small module will be overshadowed by the complexity of integration "Cost".

4.3.5 Information Hiding

It is about controlled interfaces. Modules should be specified and design so that information (algorithm and data) contained within a module is inaccessible to other modules that have no need for such information.

Hiding implies that effective modularity can be achieved by defining by a set of independent modules that communicate with one another only that information necessary to achieve software function.

The use of Information Hiding as a design criterion for modular systems provides the greatest benefits when modifications are required during testing and later, during software maintenance. Because most data and procedures are hidden from other parts of the software, inadvertent errors introduced during modifications are less likely to propagate to other location within the software.

4.3.6 Functional Independence

The concept of *functional Independence* is a direct outgrowth of modularity and the concepts of abstraction and information hiding.

Design software so that each module addresses a specific sub-function of requirements and has a simple interface when viewed from other parts of the program structure.

Functional independence is a key to good design, and design is the key to software quality.

Independence is assessed using two qualitative criteria: **cohesion** and **coupling.**

Cohesion is an indication of the relative functional strength of a module.

Coupling is an indication of the relative interdependence among modules.

A cohesive module should do just one thing.

Coupling is a qualitative indication of the degree to which a module is connected to other modules and to the outside world “lowest possible”.

4.3.7 Refinement

It is the elaboration of detail for all abstractions. It is a top down strategy.

A program is developed by successfully refining levels of procedural detail.

A hierarchy is developed by decomposing a macroscopic statement of function (a procedural abstraction) in a stepwise fashion until programming language statements are reached.

We begin with a statement of function or data that is defined at a high level of abstraction.

The statement describes function or information conceptually but provides no information about the internal workings of the function or the internal structure of the data.

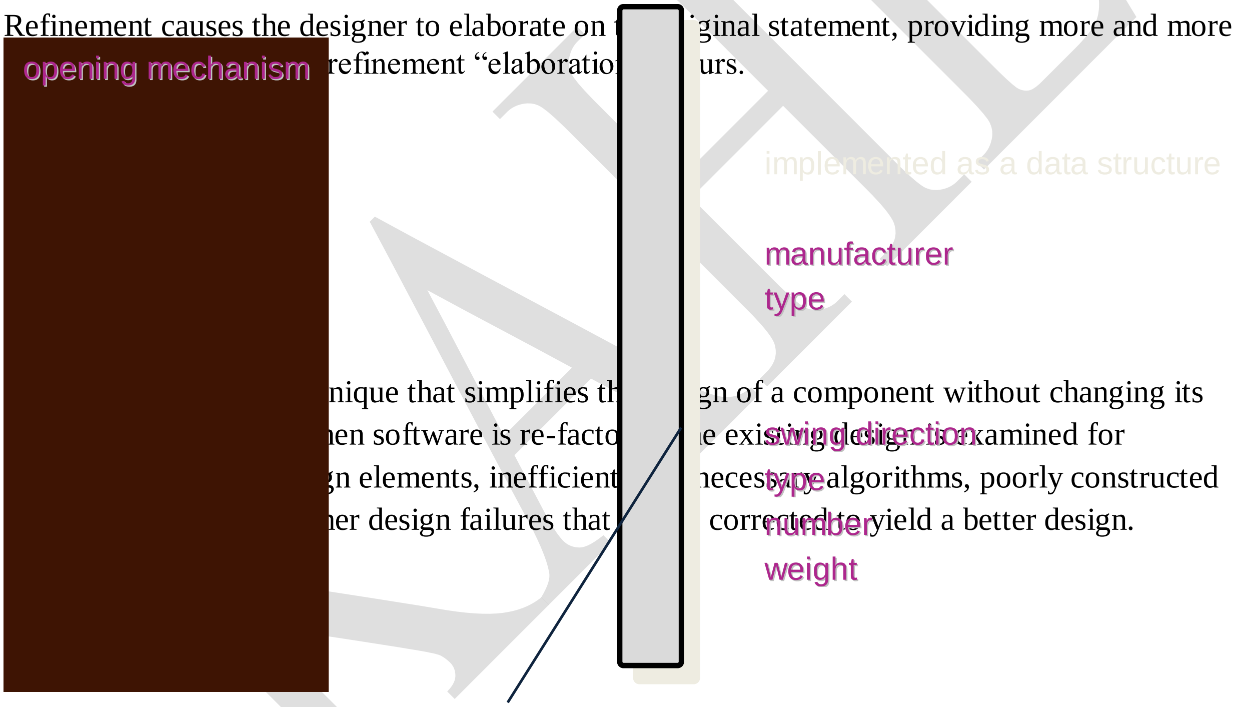
Refinement causes the designer to elaborate on the original statement, providing more and more detail as each successive refinement (elaboration) occurs.

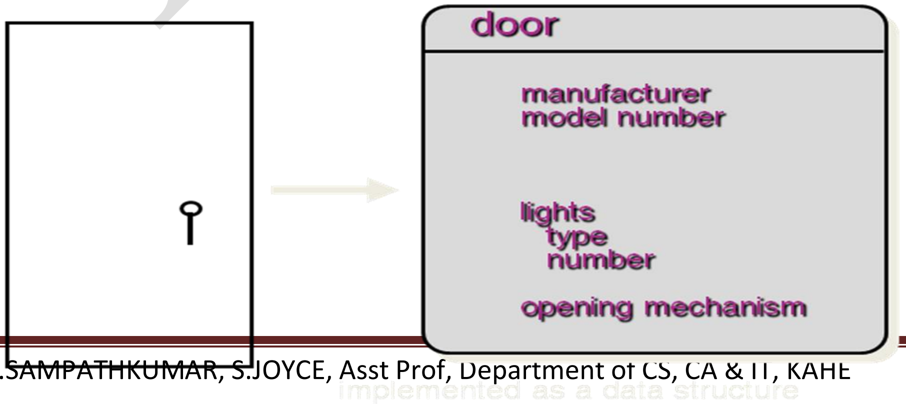
Abstraction enables a designer to specify procedure and data and yet suppress low-level details.

Refinement helps the designer to reveal low-level details as design progresses.

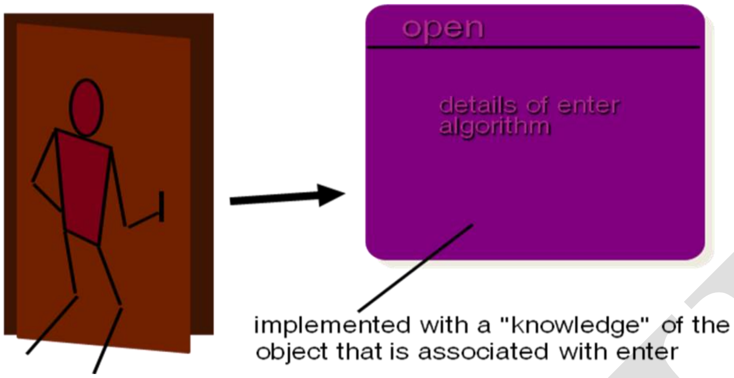
Refinement causes the designer to elaborate on t iginal statement, providing more and more
refinement “elaboratio urs.

nique that simplifies th gn of a component without changing its
hen software is re-facto e existing design is examined for
gn elements, inefficient necessary algorithms, poorly constructed
her design failures that corrected to yield a better design.





Procedural Abstraction


open
details of enter algorithm
implemented with a "knowledge" of the object that is associated with enter

"The overall structure of the software and the ways in which that structure provides conceptual integrity for a system."

- Structural properties. This aspect of the architectural design representation defines the components of a system (e.g., modules, objects, filters) and the manner in which those components are packaged and interact with one another. For example, objects are packaged to encapsulate both data and the processing that manipulates the data and interact via the invocation of methods
- Extra-functional properties. The architectural design description should address how the design architecture achieves requirements for performance, capacity, reliability, security, adaptability, and other system characteristics.
- Families of related systems. The architectural design should draw upon repeatable patterns that are commonly encountered in the design of families of similar systems. In essence, the design should have the ability to reuse architectural building blocks.

Patterns

Design Pattern Template

Pattern name—describes the essence of the pattern in a short but expressive name

Intent—describes the pattern and what it does

Also-known-as—lists any synonyms for the pattern

Motivation—provides an example of the problem

Applicability—notes specific design situations in which the pattern is applicable

Structure—describes the classes that are required to implement the pattern

Participants—describes the responsibilities of the classes that are required to implement the pattern

Collaborations—describes how the participants collaborate to carry out their responsibilities

Consequences—describes the "design forces" that affect the pattern and the potential trade-offs that must be considered when the pattern is implemented

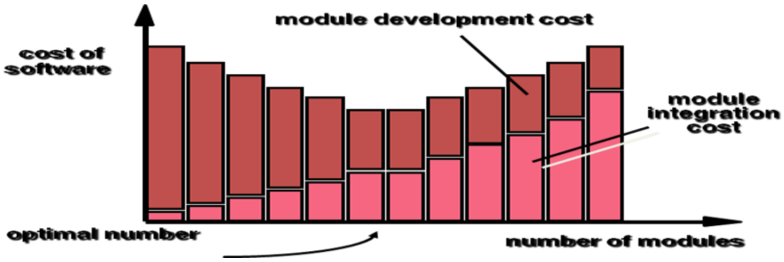
Related patterns—cross-references related design patterns

Modular Design

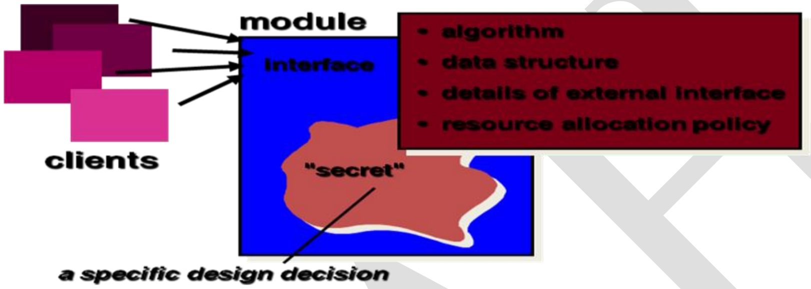
easier to build, easier to change, easier to fix ...

Modularity: Trade-offs

What is the "right" number of modules for a specific software design?



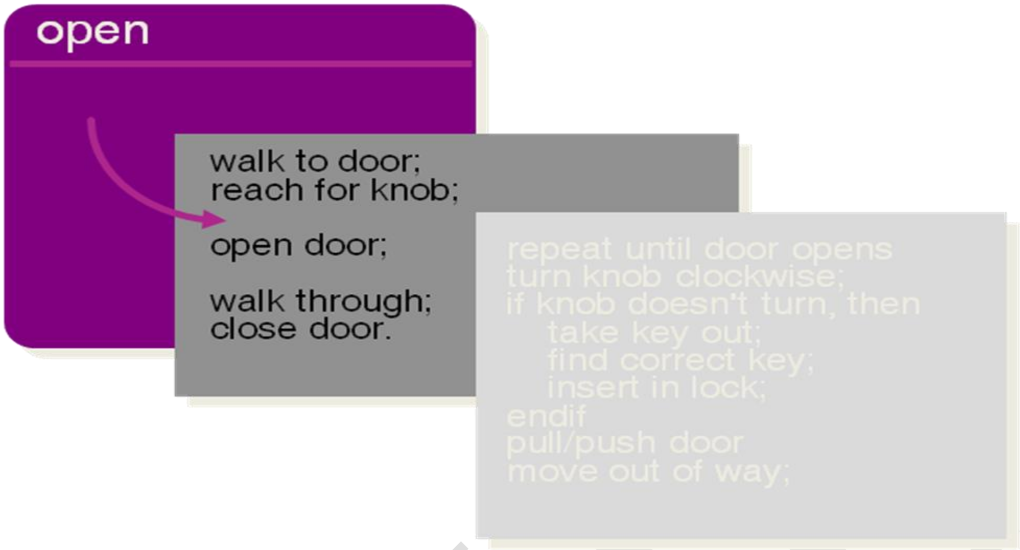
Information Hiding


module
interface
clients
algorithm
data structure
details of external interface
resource allocation policy
"secret"
a specific design decision

Why Information Hiding?

- Reduces the likelihood of "side effects"
- Limits the global impact of local design decisions
- Emphasizes communication through controlled interfaces
- Discourages the use of global data
- Leads to encapsulation—an attribute of high quality design
- Results in higher quality software

Stepwise Refinement



open

walk to door;
reach for knob;

open door;

walk through;
close door.

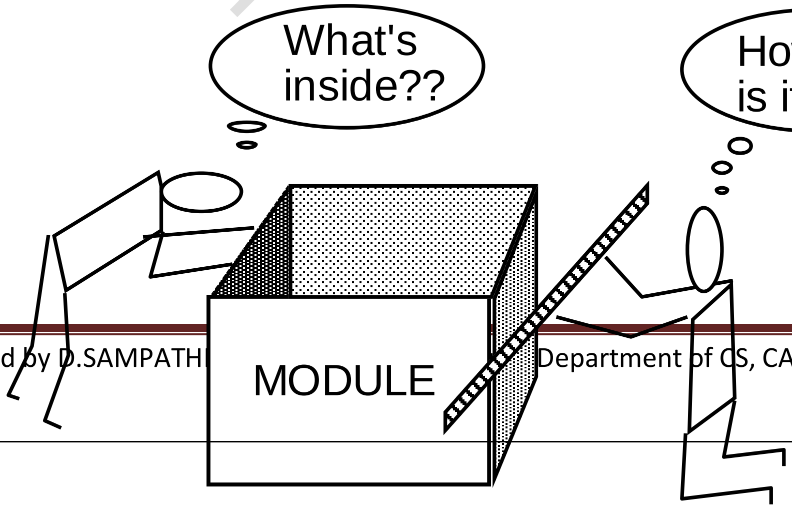
```
repeat until door opens
turn knob clockwise;
if knob doesn't turn, then
    take key out;
    find correct key;
    insert in lock;
endif
pull/push door
move out of way;
```

Functional Independence

COHESION - the degree to which a module performs one and only one function.

COUPLING - the degree to which a module is "connected" to other modules in the system.

Sizing Modules: Two Views



Refactoring

- Fowler [FOW99] defines refactoring in the following manner:
    - "Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code [design] yet improves its internal structure.”
- When software is re-factored, the existing design is examined for
    - redundancy
    - unused design elements
    - inefficient or unnecessary algorithms
    - poorly constructed or inappropriate data structures,
    - or any other design failure that can be corrected to yield a better design.

Design Concepts

- Entity classes
    - Boundary classes
    - Controller classes
- Inheritance—all responsibilities of a super-class is immediately inherited by all subclasses
- Messages—stimulate some behavior to occur in the receiving object
- Polymorphism—a characteristic that greatly reduces the effort required to extend the design

4.3.9 Design classes

As the design model evolves, the software team must define a set of *design classes* that refines the analysis classes and creates a new set of design classes.

Five different classes’ types are shown below:

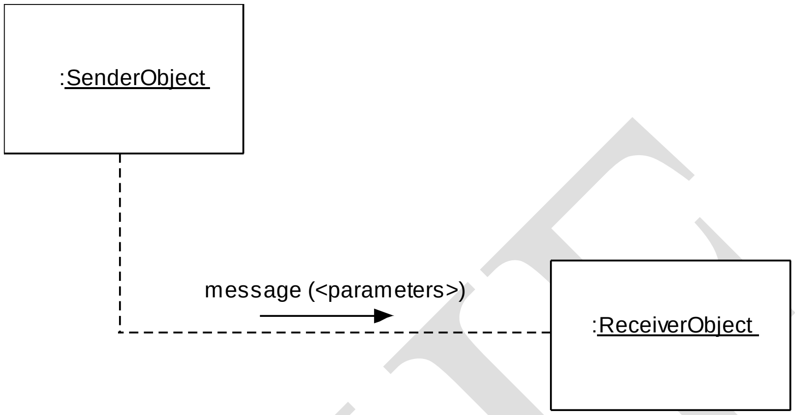
1. *User Interface classes*: define all abstractions that are necessary for HCI.
2. *Business domain classes*: are often refinements of the analysis classes defined earlier. The classes identify the attributes and services that are required to implement some element of the business domain.

3. *Process classes:* implement lower-level business abstractions required to fully manage the business domain classes.
4. *Persistent classes:* represent data stores that will persist beyond the execution of the software.
5. *System classes*: implement software management and control functions that enable the system to operate and communicate within its computing environment and with the outside world.

Inheritance (Example)

- Design 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 (a super-class) contains most of the required attributes and operations. The new class inherits from the super-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.

Messages



Polymorphism

Conventional approach …

```
case of graphtype:
        ifgraphtype = linegraph then DrawLineGraph (data);
        ifgraphtype = piechart then DrawPieChart (data);
        ifgraphtype = histogram then DrawHisto (data);
        ifgraphtype = kiviat then DrawKiviat (data);
end case;
```

All of the graphs become subclasses of a general class called graph. Using a concept called overloading [TAY90], each subclass defines an operation called *draw*. An object can send a *draw* message to any one of the objects instantiated from any one of the subclasses. The object receiving the message will invoke its own *draw* operation to create the appropriate graph.

Architectural design elements

- The architecture design elements provides us overall view of the system.
- The architectural design element is generally represented as a set of interconnected subsystem that are derived from analysis packages in the requirement model.

The architecture model is derived from following sources:

- The information about the application domain to built the software.
- Requirement model elements like data flow diagram or analysis classes, relationship and collaboration between them.
- The architectural style and pattern as per availability.

3. Interface design elements

- The interface design elements for software represents the information flow within it and out of the system.
- They communicate between the components defined as part of architecture.

Following are the important elements of the interface design:

1. The user interface
2. The external interface to the other systems, networks etc.
3. The internal interface between various components.

4. Component level diagram elements

- The component level design for software is similar to the set of detailed specification of each room in a house.
- The component level design for the software completely describes the internal details of the each software component.
- The processing of data structure occurs in a component and an interface which allows all the component operations.
- In a context of object-oriented software engineering, a component shown in a UML diagram.
- The UML diagram is used to represent the processing logic.

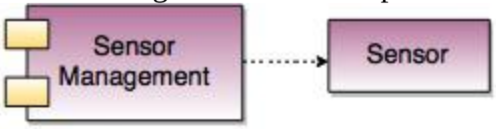


Fig. - UML component diagram for sensor managemnet

5. Deployment level design elements

- The deployment level design element shows the software functionality and subsystem that allocated in the physical computing environment which support the software.
- Following figure shows three computing environment as shown. These are the personal computer, the CPI server and the Control panel.

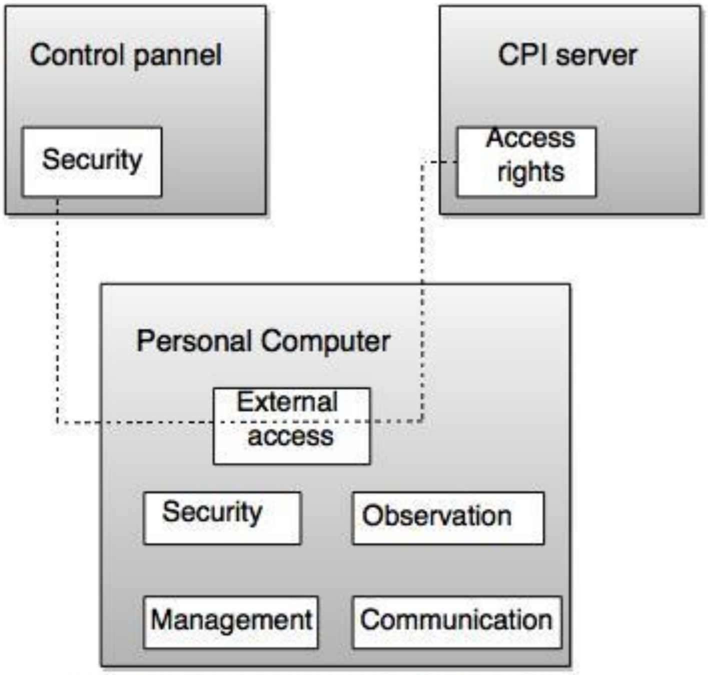


Fig. - Deployment level diagram

Software Architecture Introduction

- The concept of software architecture is similar to the architecture of building.
- The architecture is not an operational software.
- The software architecture focuses on the role of software components.
- Software components consist of a simple program module or an object oriented class in an architectural design.
- The architecture design extended and it consists of the database and the middleware that allows the configuration of a network of clients and servers.

Importance of software architecture

Following are the reasons for the importance of software architecture.

1. The representation of software architecture allows the communication between all stakeholder and the developer.
2. The architecture focuses on the early design decisions that impact on all software engineering work and it is the ultimate success of the system.
3. The software architecture composes a small and intellectually graspable model.
4. This model helps the system for integrating the components using which the components are work together.

The architectural style

- The architectural style is a transformation and it is applied to the design of an entire system.
- The main aim of architectural style is to build a structure for all components of the system.
- An architecture of the system is redefined by using the architectural style.
- An architectural pattern such as architectural style introduces a transformation on the design of an architecture.
- The software is constructed for computer based system and it shows one of the architectural style from many of style.

The design categories of architectural styles includes:

1. A set of components such as database, computational modules which perform the function required by the system.
2. A set of connectors that allows the communication, coordination and cooperation between the components.
3. The constraints which define the integration of components to form the system.
4. Semantic model allows a designer to understand the overall properties of a system by using analysis of elements.

Architectural design

- The architectural design starts then the developed software is put into the context.
- The information is obtained from the requirement model and other information collect during the requirement engineering.

Representing the system in context

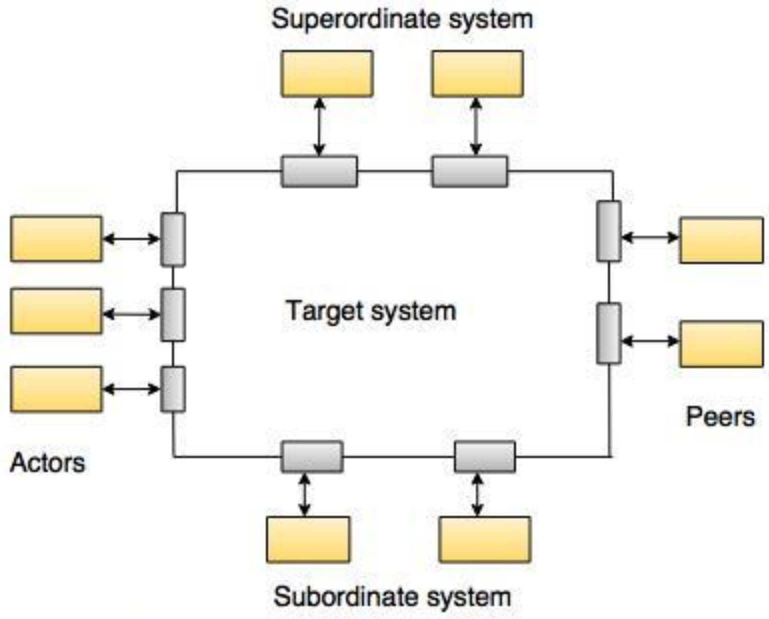


Fig. - Architectual context diagram

All the following entities communicates with the target system through the interface that is small rectangles shown in above figure.

Superordinate system
These system use the target system like a part of some higher-level processing scheme.

Subordinate system
This systems is used by the target system and provide the data mandatory to complete target system functionality.

Peer-level system
These system interact on peer-to-peer basis means the information is consumed by the target system and the peers.

Actors
These are the entities like people, device which interact with the target system by consuming information that is mandatory for requisite processing.

Defining Archetype

- An archetype is a class or pattern which represents a core abstraction i.e critical to implement or design for the target system.
- A small set of archetype is needed to design even the systems are relatively complex.
- The target system consists of archetype that represent the stable elements of the architecture.
- Archetype is instantiated in many different forms based on the behavior of the system.
- In many cases, the archetype is obtained by examining the analysis of classes defined as a part of the requirement model.

An Architecture Trade-off Analysis Method (ATAM)

ATAM was developed by the Software Engineering Institute (SEI) which started an iterative evaluation process for software architecture.

The design analysis activities which are executed iteratively that are as follows:

1. Collect framework
Collect framework developed a set of use cases that represent the system according to user point of view.

2. Obtained requirement, Constraints, description of the environment.
These types of information are found as a part of requirement engineering and is used to verify all the stakeholders are addressed properly.

3. Describe the architectural pattern
The architectural patterns are described using an architectural views which are as follows:

Module view: This view is for the analysis of assignment work with the components and the degree in which abstraction or information hiding is achieved

Process view: This view is for the analysis of the software or system performance.

Data flow view: This view analyzes the level and check whether functional requirements are met to the architecture.

4. Consider the quality attribute in segregation
The quality attributes for architectural design consist of reliability, performance, security, maintainability, flexibility, testability, portability, re-usability etc.

5. Identify the quality attributes sensitivity

- The sensitivity of quality attributes achieved by making the small changes in the architecture and find the sensitivity of the quality attribute which affects the performance.
- The attributes affected by the variation in the architecture are known as sensitivity points.

Data Design at the Architectural Level and Component Level

The data design action translates data defined as part of the analysis model into data structures at the software component level and. When necessary into a database architecture at the application level.

a) Data Design at the Architectural Level

The challenge in data design is to extract useful information from this data environment, particularly when the information desired is cross-functional.

To solve this challenge, the business IT community has developed data mining techniques, also called knowledge discovery in database (KDD) , that navigate through existing databases in an attempt to extract appropriate business-level information. An alternative solution, called a data warehouse, adds an additional layer to the data architecture.

A data warehouse is a separate data environment that is not directly integrated with day –to-day application but encompasses all data used by a business.

b) Data Design at the Component Level

Data design at the component level focuses on the representation of the data structures that are directly accessed by one or more software components. We consider the following set of principles (adapted from for data specification):

1. The systematic analysis principles applied to function and behavior should also be applied to data.
2. All data structure and the operations to be performed on each should be identified.
3. A mechanism for defining the content of each data object should be established and used to define both data and the operation applied it.
4. Low-level design decision should be known only to those modules that must make direct use of the data contained within the structure.
5. The representation of a data structure should be known only to those modules that must make direct use of the data contained within the structure.
6. A library of useful data structures and the operations that may be applied to them should be developed.
7. A software design and programming language should support the specification and realization of abstract data types.

Mapping Data Flow into Software Architecture

This section describes the general process of mapping requirements into software architectures during the structured design process. The technique described in this chapter is based on analysis of the data flow diagram discussed in Chapter 8.

An Architectural Design Method

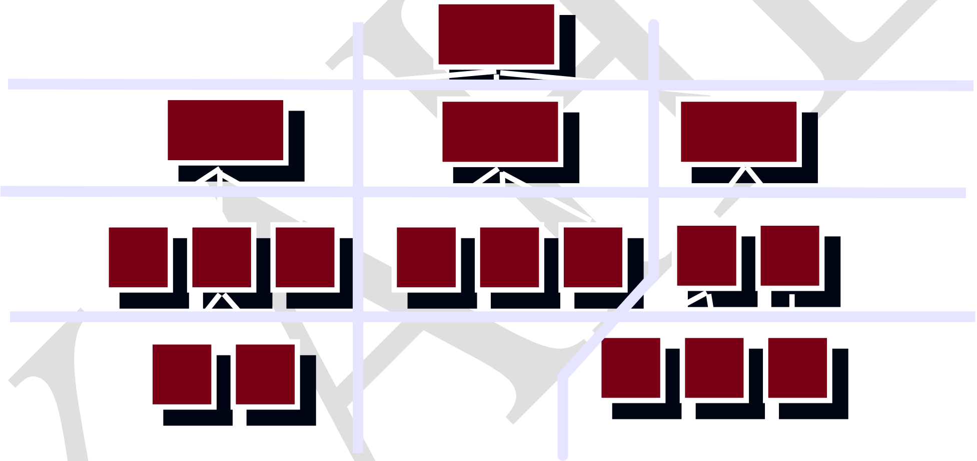
customer requirements

four bedrooms, three baths, lots of glass…

Deriving Program Architecture

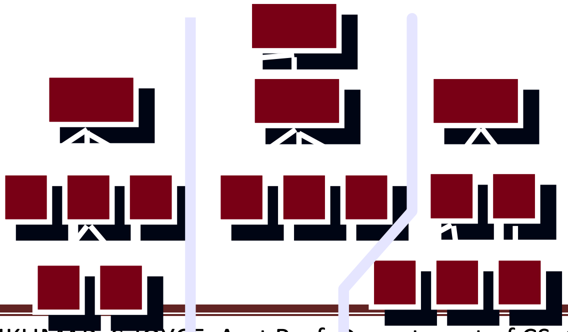
Partitioning the Architecture

horizontal" and "vertical" partitioning are required



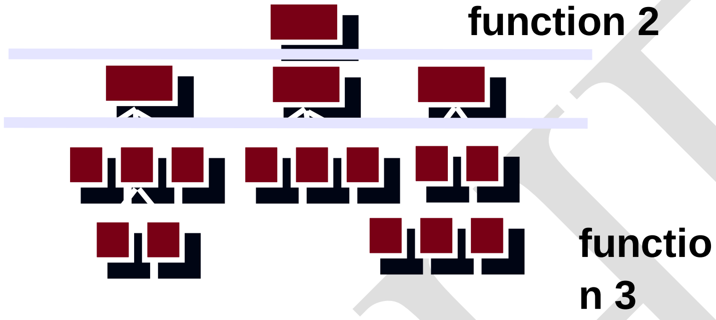
Horizontal Partitioning

- define separate branches of the module hierarchy for each major function
- use control modules to coordinate communication between functions



Vertical Partitioning:
Factoring

- design so that decision making and work are stratified
- decision making modules should reside at the top of the architecture



Why Partitioned Architecture?

- results in software that is easier to test
- leads to software that is easier to maintain
- results in propagation of fewer side effects
- results in software that is easier to extend

- objective: to derive a program architecture that is partitioned
- approach:
  - the DFD is mapped into a program architecture
  - the PSPEC and STD are used to indicate the content of each module
- notation: structure chart

Flow Characteristics

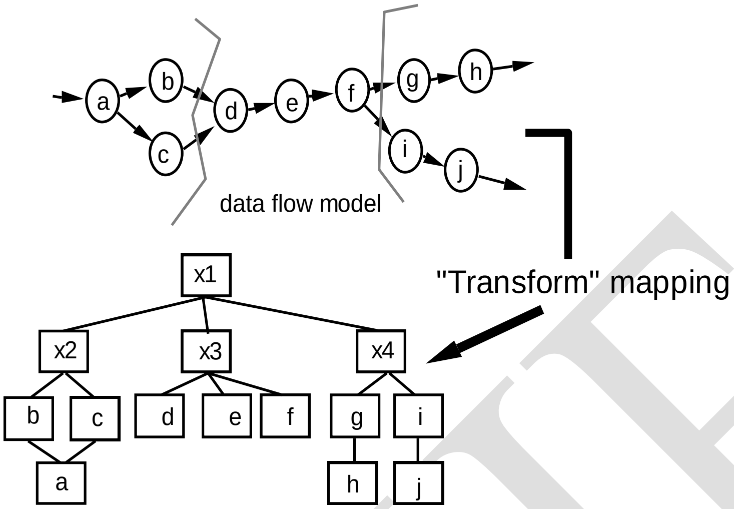
General Mapping Approach

Isolate incoming and outgoing flow boundaries; for transaction flows, isolate the transaction center.

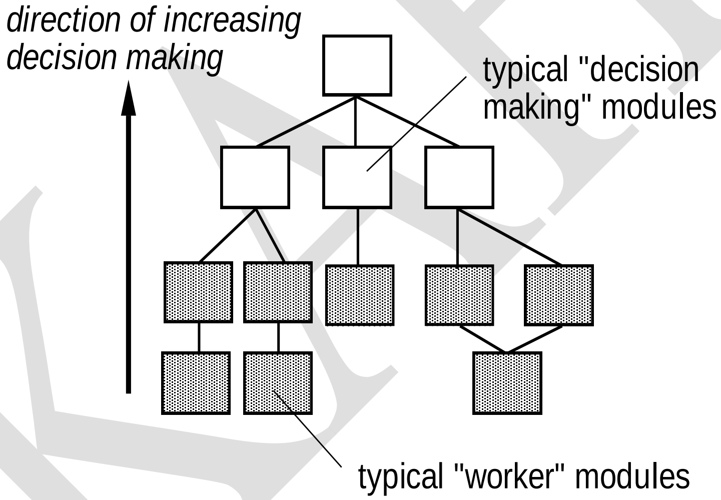
Working from the boundary outward, mapDFD transforms into corresponding modules.

Add control modules as required.

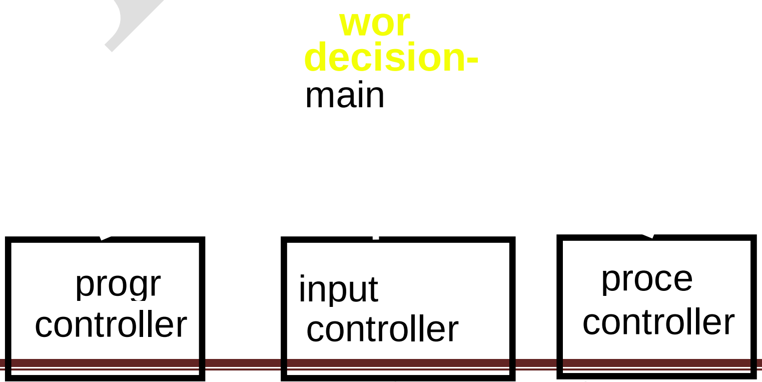
Refine the resultant program structureusing effective modularity concepts.



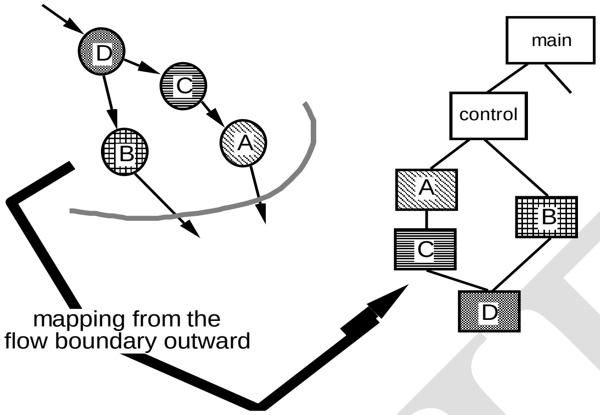
Factoring


direction of increasing decision making
typical "decision making" modules
typical "worker" modules

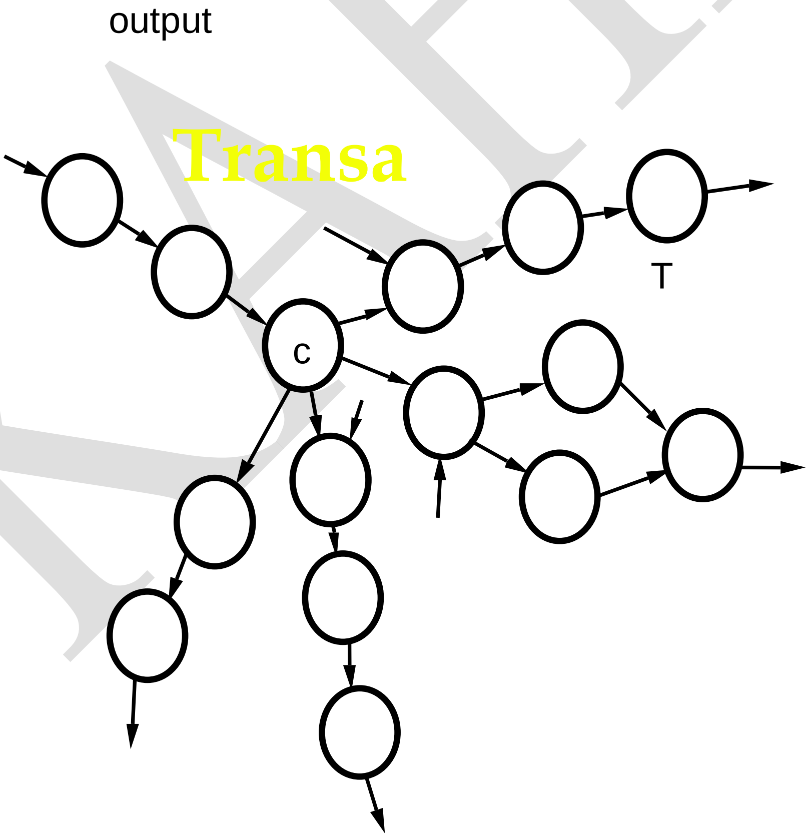
First Level Factoring



Second Level Mapping



Transaction Flow



Refining the Analysis Model

1. Write an English language processing narrative for the level 01 flow model
2. Apply noun/verb parse to isolate processes, data items, store and entities
3. Develop level 02 and 03 flow models
4. Create corresponding data dictionary entries
5. Refine flow models as appropriate

Modeling Component Level Design

Overview

The purpose of component-level design is to define data structures, algorithms, interface characteristics, and communication mechanisms for each software component identified in the architectural design. Component-level design occurs after the data and architectural designs are established. The component-level design represents the software in a way that allows the designer to review it for correctness and consistency, before it is built. The work product produced is a design for each software component, represented using graphical, tabular, or text-based notation. Design walkthroughs are conducted to determine correctness of the data transformation or control transformation allocated to each component during earlier design steps.

Component Definitions

- Component is a modular, deployable, replaceable part of a system that encapsulates implementation and exposes a set of interfaces
- Object-oriented view is that component contains a set of collaborating classes
    - Each elaborated class includes all attributes and operations relevant to its implementation
    - All interfaces communication and collaboration with other design classes are also defined
    - Analysis classes and infrastructure classes serve as the basis for object-oriented elaboration
- Traditional view is that a component (or module) reside in the software and serves one of three roles
    - Control components coordinate invocation of all other problem domain components
    - Problem domain components implement a function required by the customer
    - Infrastructure components are responsible for functions needed to support the processing required in a domain application
    - The analysis model data flow diagram is mapped into a module hierarchy as the starting point for the component derivation

- Process-Related view emphasizes building systems out of existing components chosen from a catalog of reusable components as a means of populating the architecture

Class-based Component Design

- Focuses on the elaboration of domain specific analysis classes and the definition of infrastructure classes
- Detailed description of class attributes, operations, and interfaces is required prior to beginning construction activities

Class-based Component Design Principles

- Open-Closed Principle (OCP) – class should be open for extension but closed for modification
- Liskov Substitution Principle (LSP) – subclasses should be substitutable for their base classes
- Dependency Inversion Principle (DIP) – depend on abstractions, do not depend on concretions
- Interface Segregation Principle (ISP) – many client specific interfaces are better than one general purpose interface
- Release Reuse Equivalency Principle (REP) – the granule of reuse is the granule of release
- Common Closure Principle (CCP) – classes that change together belong together
- Common Reuse Principle (CRP) – Classes that can’t be used together should not be grouped together

Component-Level Design Guidelines

- Components
  - Establish naming conventions in during architectural modeling
  - Architectural component names should have meaning to stakeholders
  - Infrastructure component names should reflect implementation specific meanings
  - Use of stereotypes may help identify the nature of components
- Interfaces
  - Use lollipop representation rather than formal UML box and arrow notation
  - For consistency interfaces should flow from the left-hand side of the component box
  - Show only the interfaces relevant to the component under construction
- Dependencies and Inheritance
  - For improved readability model dependencies from left to right and inheritance from bottom (derived classes) to top (base classes)
  - Component interdependencies should be represented by interfaces rather that component to component dependencies

Cohesion (lowest to highest)

- Utility cohesion – components grouped within the same category but are otherwise unrelated
- Temporal cohesion – operations are performed to reflect a specific behavior or state
- Procedural cohesion – components grouped to allow one be invoked immediately after the preceding one was invoked with or without passing data
- Communicational cohesion –operations required same data are grouped in same class
- Sequential cohesion – components grouped to allow input to be passed from first to second and so on
- Layer cohesion – exhibited by package components when a higher level layer accesses the services of a lower layer, but lower level layers do not access higher level layer services

- Functional cohesion – module performs one and only one function

Coupling

- Content coupling – occurs when one component surreptitiously modifies internal data in another component
- Common coupling – occurs when several components make use of a global variable
- Control coupling – occurs when one component passes control flags as arguments to another
- Stamp coupling – occurs when parts of larger data structures are passed between components
- Data coupling – occurs when long strings of arguments are passed between components
- Routine call coupling – occurs when one operator invokes another
- Type use coupling – occurs when one component uses a data type defined in another

- Inclusion or import coupling – occurs when one component imports a package or uses the content of another

- External coupling – occurs when a components communications or collaborates with infrastructure components (e.g. database)

Conducting Component-Level Design

1. Identify all design classes that correspond to the problem domain.
2. Identify all design classes that correspond to the infrastructure domain.
3. Elaborate all design classes that are not acquired as reusable components.
   a. Specify message details when classes or components collaborate.
   b. Identify appropriate interfaces for each component.
   c. Elaborate attributes and define data types and data structures required to implement them.
   d. Describe processing flow within each operation in detail.
4. Identify persistent data sources (databases and files) and identify the classes required to manage them.
5. Develop and elaborate behavioral representations for each class or component.

6. Elaborate deployment diagrams to provide additional implementation detail.
7. Refactor every component-level diagram representation and consider alternatives.

WebApp Component-Level Design

- Boundary between content and function often blurred
- WebApp component is defined is either a:
    - well-defined cohesive function manipulates content or provides computational or data processing for an end- user or
    - cohesive package of content and functionality that provides the end-user with some required capability

WebApp Component-Level Content Design

- Focuses on content objects and the manner in which they may be packaged for presentation to the end-user
- As the WebApp size increases so does the need for formal representations and easy content reference and manipulation
- For highly dynamic content a clear structural model incorporating content components should be established

WepApp Component-Level Functional Design

- WebApps provide sophisticated processing functions
    - perform dynamic processing to create content and navigational capability
    - provide business domain appropriate computation or data processing
    - provide database query and access
    - establish interfaces with external corporate systems
- WebApp functionality is delivered as a series of components developed in parallel
- During architectural design WebApp content and functionality are combined to create a functional architecture
- The functional architecture is a representation of the functional domain of the WebApp and describes how the components interact with each other

Traditional Component-Level Design

- Each block of code has a single entry at the top
- Each block of code has a single exit at the bottom
- Only three control structures are required: sequence, condition (if-then-else), and repetition (looping)
- Reduces program complexity by enhancing readability, testability, and maintainability

Design Notation

- Graphical
  - UML activity diagrams
  - Flowcharts – arrows for flow of control, diamonds for decisions, rectangles for processes
- Tabular
  - Decision table – subsets of system conditions and actions are associated with each other to define the rules for processing inputs and events
- Program Design Language (PDL)
  - Structured English or pseudocode used to describe processing details
  - Fixed syntax with keywords providing for representation of all structured constructs, data declarations, and module definitions
  - Free syntax of natural language for describing processing features
  - Data declaration facilities for simple and complex data structures
  - Subprogram definition and invocation facilities

Component-Based Development

- CBSE is a process that emphasizes the design and construction of computer-based systems from a catalog of reusable software components
- CBSE is a time and cost effective
- Requires software engineers to reuse rather than reinvent
- Management can be convinced to incur the additional expense required to create reusable components by amortizing the cost over multiple projects
- Libraries can be created to make reusable components easy to locate and easy to incorporate them in new systems

Possible Questions

Part – B (2 Mark)

1. Define abstraction.
2. Differentiate between refinement and refactoring
3. Write the difference between transform flow and transaction flow
4. What is transform mapping?
5. Define transaction mapping.

Part – C (6 Mark)

1. Explain in detail the process of data design at
    a. Architectural level ii) Component level
2. Write in detail the approach used to design class based components
3. Discuss in detail about the Architectural components of software.
4. Write short notes on
    a. Transform mapping ii) Transaction mapping
5. Write short notes on the following design concepts
    a. Information hiding ii) Refinement iii) Refactoring
6. Describe in detail the procedure to refine an architecture into components.
7. Write short notes on the following design concepts
    a. Abstraction ii) Architecture iii) Modularity
8. Write in detail the approach used to design conventional components
9. Explain in detail about design process and design quality
10. Write short notes on
    a. Transform flow ii) Transaction flow

KARPAGAM ACADEMY OF HIGHER EDUCATION

DEPARTMENT OF COMPUTER SCIENCE

II BSc CS BATCH(2016 - 2019)

PART-A OBJECTIVE TYPE/ MULTIPLE CHOICE QUESTIONS

UNIT-4

S.NO	Questions	Opt1	Opt2	Opt3	Opt4	Answer
1	There are __________ major phases to any design process	2	3	4	5	2
2	Diversification is the ____________ of a repertoire of alternatives.	component	solution	acquisition	knowledge	acquisition
3	During ____________, the designer chooses and combines appropriate elements from the	diversification	convergence	elimination	creation	convergence
4	________ and __________ combine intuition and judgement based on experience in building	elimination, convergence	creation, convergence	acquisition, creation	diversification and convergence	diversification and convergence
5	__________ can be traced to a customer's requirements and at the same time assessed for quality	design	analysis	principles	testing	design
6	The __________ must implement all of the explicit requirements contained in the analysis model	principles	testing	design	component	design
7	A ___________ should exhibit an architectural structure that has been created using recognizable	principles	testing	component	design	design
8	A ___________ is composed of components that exhibit good design characteristics.	principles	testing	component	design	design
9	A ___________ can be implemented in an evolutionary fashion thereby facilitating	principles	testing	component	design	design
10	A ___________ should be modular that is the software should be logically partitioned into	design	principles	component	testing	design
11	A ___________ should contain distinct representations of data, architecture, interfaces, and	design	principles	component	testing	design
12	A ___________ should lead to data structures that are appropriate for the objects to be implemented	design	principles	component	testing	design
13	. A _____________ should lead to interfaces that reduce the complexity of connections	design	principles	component	testing	design
14	A ___________ should be derived using a repeatable method that is driven by information	principles	component	design	testing	design
15	The software __________ process encourages good design through the application of fundamental	principles	component	design	testing	design
16	The __________ must be a readable, understandable guide for those who generate code and for	principles	component	design	testing	design

17	The __________ should provide a complete picture of the software addressing the data, functional and	principles	component	design	testing	design
18	The evolution of software __________ is a continuing process that has spanned the past	principles	component	design	testing	design
19	Procedural aspects of design definition evolved into a philosophy called ____________.	top down programming	bottom up programming	structured programming	object oriented programming	structured programming
20	The design process should not suffer from ___________.	analysis	tunnel vision	conceptual errors	integrity	tunnel vision
21	The design should be __________ to the analysis model.	consistent	related	traceable	relevant	traceable
22	The design should not ___________ the wheel.	minimize	maximize	integrate	reinvent	reinvent
23	The design should ___________ the intellectual distance	maximize	minimize	integrate	analyse	minimize
24	. The ___________ is represented at a high level of abstraction	specification	analysis	quality	design specification	design specification
25	The design should exhibit ___________ and integration.	uniformity	analysis	quality	review	uniformity
26	The design should be ____________ to accommodate change.	reviewed	analysed	assessed	structured	structured
27	The design should be ___________ to degrade gently, even when aberrant data, events,	reviewed	analysed	assessed	structured	structured
28	Design is not ___________, coding is not design	coding	analysis	review	event	coding
29	Design is not coding, __________ is not design.	coding	analysis	review	event	coding
30	The design should be __________ for quality as it is being created not after the fact.	reviewed	assessed	structured	integrated	assessed
31	The design should be ___________ to minimize conceptual errors.	reviewed	assessed	structured	integrated	reviewed
32	Software design is both a _________ and a model.	model	process	data	function	process
33	__________ is the only way that we can accurately translate a customer's requirements into a	specification	design	data	prototype	design
34	The design ___________ is the equivalent of an architect's plan for a house.	analysis	process	model	function	model
35	At the highest level of _________, a solution is stated in broad terms, using the language of the problem	refinement	modularity	abstraction	continuity	abstraction
36	. A __________ is a named sequence of instructions that has a specific and limited function.	procedural abstraction	data abstraction	control abstraction	Process abstraction	procedural abstraction
37	A __________ is a named collection of data that describes a data object.	procedural abstraction	data abstraction	control abstraction	Process abstraction	data abstraction
38	_________ implies a program control mechanism without specifying internal detail.	procedural abstraction	data abstraction	control abstraction	Process abstraction	control abstraction

39	___________ is used to coordinate activities in an operating system.	synchronization semaphore	control abstraction	data abstraction	procedural abstraction	synchronization semaphore
40	_________ is a top down design strategy originally proposed by Niklaus Wirth.	stepwise refinement	control abstraction	data abstraction	procedural abstraction	stepwise refinement
41	The designer's goal is to produce a model or representation of a __________ that will later be built	component	entity	data	raw material	component
42	The second phase of any design process is the gradual ___________ of all but one	acquisition	addition	elimination	creation	elimination
43	Design begins with the __________ model.	data	requirements	specification	code	requirements
44	Software design methodologies lack the __________ that are normally associated with more	depth	flexibility	quantitative nature	all of the above	all of the above
45	Software requirements, manifested by the ___________ models, feed the design task.	data	functional	behavioral	all of the above	all of the above
46	___________ is the place where quality is fostered in software engineering	model	data	design	specification	design
47	________ provides us with representations of software that can be assessed for quality.	design	specification	data	prototype	design
48	Procedural aspects of design definition evolved into a philosophy called ___________.	procedural programming	object oriented programming	structured programming	all of the above	structured programming
49	Meyer defines __________ criteria that enable us to evaluate a design method with respect to its	2	3	4	5	5
50	. If a design method provides a systematic mechanism for decomposing the problem into sub	modular decomposability	modular composability	modular understandability	modular continuity	modular decomposability
51	If a design method enables existing (reusable) design components to be assembled into a	modular decomposability	modular composability	modular understandability	modular continuity	modular composability
52	If a module can be understood as a stand alone unit (without reference to other modules), it will be easier	modular decomposability	modular composability	modular understandability	modular continuity	modular understandability
53	If small changes to the system requirements result in changes to individual modules, rather than	modular decomposability	modular composability	modular understandability	modular continuity	modular continuity
54	If an aberrant condition occurs within a module and its effects are constrained within that module,	modular protection	modular composability	modular understandability	modular continuity	modular protection
55	The aspect of the architectural design representation defines the components of a system and the	extra functional property	structural property	families of related systems	none of the above	structural property
56	____________ represent architecture as an organized collection of program components.	dynamic models	functional models	framework models	structural models	structural models
57	____________ increases the level of design abstraction by attempting to identity repeatable	framework models	dynamic models	process models	functional models	framework models
58	_________ address the behavioural aspects of the program architecture, indicating	framework models	dynamic models	process models	functional models	dynamic models
59	___________ focus on the design of the business or technical process that the system must	framework models	dynamic models	process models	functional models	process models
60	_____________ can be used to represent the functional hierarchy of a system.	framework models	dynamic models	process models	functional models	functional models

UNIT-V

SYLLABUS

Testing Strategies & Tactics: Software Testing Fundamentals, Strategic Approach to Software Testing, Test Strategies for Conventional Software, Validation Testing, System testing Black-Box Testing, White-Box Testing and their type, Basis Path Testing

SOFTWARE TESTING FUNDAMENTALS:

- Testing presents an interesting anomaly for the software engineer. During earlier software engineering activities, the engineer attempts to build software from an abstract concept to a tangible product.
- The engineer creates a series of test cases that are intended to "demolish" the software that has been built.
- In fact, testing is the one step in the software process that could be viewed (psychologically, at least) as destructive rather than constructive.
- Software engineers are by their nature constructive people.
- Testing requires that the developer discard preconceived notions of the "correctness" of software just developed and overcome a conflict of interest that occurs when errors are uncovered.
- Beizer describes this situation effectively when he states: There's a myth that if we were really good at programming, there would be no bugs to catch. If only we could really concentrate, if only everyone used structured programming, top down design, decision tables, if programs were written in SQUISH, if we had the right silver bullets, then there would be no bugs. So goes the myth. There are bugs, the myth says, because we are bad at what we do; and if we are bad at it, we should feel guilty about it. Therefore, testing and test case design is an admission of failure, which instills a goodly dose of guilt.

Testing Objectives

Glen Myers states a number of rules that can serve well as testing objectives:

1. Testing is a process of executing a program with the intent of finding an error.

2. A good test case is one that has a high probability of finding an as-yet undiscovered error.

3. A successful test is one that uncovers an as-yet-undiscovered error.

If testing is conducted successfully (according to the objectives stated previously), it will uncover errors in the software.

Also testing demonstrates that software functions appear to be working according to specification, that behavioral and performance requirements appear to have been met.

In addition, data collected as testing is conducted provide a good indication of software reliability and some indication of software quality as a whole.

But testing cannot show the absence of errors and defects, it can show only that software errors and defects are present.

Testing Principles

Before applying methods to design effective test cases, a software engineer must understand the basic principles that guide software testing. Davis [DAV95] suggests a set of testing principles.

All tests should be traceable to customer requirements.

The objective of software testing is to uncover errors. It follows that the most severe defects (from the customer's point of view) are those that cause the program to fail to meet its requirements.

Tests should be planned long before testing begins.

Test planning can begin as soon as the requirements model is complete.

Detailed definition of test cases can begin as soon as the design model has been solidified. Therefore, all tests can be planned and designed before any code has been generated.

The Pareto principle applies to software testing.

Pareto principle implies that 80 percent of all errors uncovered during testing will likely be traceable to 20 percent of all program components. The problem, of course, is to isolate these suspect components and to thoroughly test them.

Testing should begin "in the small" and progress toward testing "in the large."

The first tests planned and executed generally focus on individual components. As testing progresses, focus shifts in an attempt to find errors in integrated clusters of components and ultimately in the entire system.

Exhaustive testing is not possible.

The number of path permutations for even a moderately sized program is exceptionally large. For this reason, it is impossible to execute every combination of paths during testing. It is possible, however, to adequately cover program logic and to ensure that all conditions in the component-level design have been exercised.

To be most effective, testing should be conducted by an independent third party.

Testability

- Software testability is simply how easily a computer program can be tested.
- Since testing is so profoundly difficult, it pays to know what can be done to streamline it.
- Sometimes programmers are willing to do things that will help the testing process and a checklist of possible design points, features, etc., can be useful in negotiating with them.
- "Testability" occurs as a result of good design. Data design, architecture, interfaces, and component-level detail can either facilitate testing or make it difficult.

The **checklist** that follows provides a set of characteristics that lead to testable software.

Operability. "The better it works, the more efficiently it can be tested."

• The system has few bugs (bugs add analysis and reporting overhead to the test process).

• No bugs block the execution of tests.

• The product evolves in functional stages (allows simultaneous development and testing).

Observability. "What you see is what you test."

• Distinct output is generated for each input.

• System states and variables are visible or queriable during execution.

• Past system states and variables are visible or queriable (e.g., transaction logs).

• All factors affecting the output are visible.

• Incorrect output is easily identified.

• Internal errors are automatically detected through self-testing mechanisms.

• Internal errors are automatically reported.

• Source code is accessible.

Controllability. "The better we can control the software, the more the testing can be automated and optimized."

• All possible outputs can be generated through some combination of input.

• All code is executable through some combination of input.

• Software and hardware states and variables can be controlled directly by the test engineer.

• Input and output formats are consistent and structured.

• Tests can be conveniently specified, automated, and reproduced.

Decomposability. "By controlling the scope of testing, we can more quickly isolate problems and perform smarter retesting."

• The software system is built from independent modules.

• Software modules can be tested independently.

Simplicity. "The less there is to test, the more quickly we can test it."

• Functional simplicity (e.g., the feature set is the minimum necessary to meet requirements).

• Structural simplicity (e.g., architecture is modularized to limit the propagation of faults).

• Code simplicity (e.g., a coding standard is adopted for ease of inspection and maintenance).

Stability. "The fewer the changes, the fewer the disruptions to testing."

• Changes to the software are infrequent.

- Changes to the software are controlled.
- Changes to the software do not invalidate existing tests.
- The software recovers well from failures.

Understandability. "The more information we have, the smarter we will test."

- The design is well understood.
- Dependencies between internal, external, and shared components are well understood.
- Changes to the design are communicated.
- Technical documentation is instantly accessible.
- Technical documentation is well organized.
- Technical documentation is specific and detailed.
- Technical documentation is accurate.

Kaner, Falk, and Nguyen suggest the following **attributes of a "good" test**:

1. **A good test has a high probability of finding an error.**

- To achieve this goal, the tester must understand the software and attempt to develop a mental picture of how the software might fail.
- Ideally, the classes of failure are probed. For example, one class of potential failure in a GUI (graphical user interface) is a failure to recognize proper mouse position.
- A set of tests would be designed to exercise the mouse in an attempt to demonstrate an error in mouse position recognition.

2. **A good test is not redundant.**

- Testing time and resources are limited. There is no point in conducting a test that has the same purpose as another test. Every test should have a different purpose.

3. A good test should be "best of breed".

- In a group of tests that have a similar intent, time and resource limitations may mitigate toward the execution of only a subset of these tests.
- In such cases, the test that has the highest likelihood of uncovering a whole class of errors should be used.

4. A good test should be neither too simple nor too complex.

- Although it is sometimes possible to combine a series of tests into one test case, the possible side effects associated with this approach may mask errors.
- In general, each test should be executed separately.

A STRATEGIC APPROACH TO SOFTWARE TESTING

Testing is a set of activities that can be planned in advance and conducted systematically. For this reason a template for software testing—a set of steps into which we can place specific test-case design techniques and testing methods—should be defined for the software process.

A strategy for software testing must accommodate low-level tests that are necessary to verify that a small source code segment has been correctly implemented as well as high-level tests that validate major system functions against customer requirements. A strategy should provide guidance for the practitioner and a set of milestones for the manager. Because the steps of the test strategy

occur at a time when deadline pressure begins to rise, progress must be measurable and problems should surface as early as possible.

Characteristics:

- To perform effective testing, you should conduct effective technical reviews. By doing this, many errors will be eliminated before testing commences.
- Testing begins at the component level and works "outward" toward the integration of the entire computer-based system.
- Different testing techniques are appropriate for different software engineering approaches and at different points in time.
- Testing is conducted by the developer of the software and (for large projects) an independent test group.
- Testing and debugging are different activities, but debugging must be accommodated in any testing strategy.

TEST STRATEGIES FOR CONVENTIONAL SOFTWARE

VALIDATION TESTING

Validation testing begins at the culmination of integration testing, when individual components have been exercised, the software is completely assembled as a package, and interfacing errors have been uncovered and corrected. At the validation or system level, the distinction between different software categories disappears. Testing focuses on user-visible actions and user-recognizable output from the system.

Validation can be defined in many ways, but a simple (albeit harsh) definition is that validation succeeds when software functions in a manner that can be reasonably expected by the customer. At this point a battle-hardened software developer might protest: "Who or what is the arbiter of reasonable expectations?" If a Software Requirements Specification has been developed, it describes all user-visible attributes of the software and contains a Validation Criteria section that forms the basis for a validation-testing approach.

i) Validation-Test Criteria

Software validation is achieved through a series of tests that demonstrate conformity with requirements. A test plan outlines the classes of tests to be conducted, and a test procedure defines specific test cases that are designed to ensure that all functional requirements are satisfied, all behavioral characteristics are achieved, all content is accurate and properly presented, all performance requirements are attained, documentation is correct, and usability and other requirements are met (e.g., transportability, compatibility, error recovery, maintainability). If a deviation from specification is uncovered, a deficiency list is created. A method for resolving deficiencies (acceptable to stakeholders) must be established.

ii) Configuration Review

An important element of the validation process is a configuration review. The intent of the review is to ensure that all elements of the software configuration have been properly developed, are cataloged, and have the necessary detail to bolster the support activities. The configuration review, sometimes called an audit.

ii) Alpha and Beta Testing

It is virtually impossible for a software developer to foresee how the customerwill really use a program. Instructions for use may be misinterpreted; strange combinations of data may be used; output that seemed clear to the tester may be unintelligible to a user in the field. When custom software is built for one customer, a series of acceptance tests are conducted to enable the customer to validate all requirements. Conducted by the end user rather than software engineers, an acceptance test can range from an informal “test drive” to a planned and systematically executed series of tests. In fact, acceptance testing can be conducted over a period of weeks or months, thereby uncovering cumulative errors that might degrade the system over time. If software is developed as a product to be used by many customers, it is impractical to perform formal acceptance tests with each one. Most softwareLike all other testing steps, validation tries to uncover errors, but the focus is at the requirements level—on things that will be immediately apparent to the end user. product builders use a process called alpha and beta testing to uncover errors that only the end user seems able to find.

The alpha test is conducted at the developer’s site by a representative group of end users. The software is used in a natural setting with the developer “looking over the shoulder” of the users and recording errors and usage problems. Alpha tests are conducted in a controlled environment.

The beta test is conducted at one or more end-user sites. Unlike alpha testing, the developer generally is not present. Therefore, the beta test is a “live” application of the software in an environment that cannot be controlled by the developer. The customer records all problems (real or imagined) that are encountered during beta testing and reports these to the developer at regular intervals. As a result of problems reported during beta tests, you make modifications and then prepare for release of the software product to the entire customer base. A variation on beta testing, called customer acceptance testing, is sometimes performed when custom software is delivered to a customer under contract.

The customer performs a series of specific tests in an attempt to uncover errors before accepting the software from the developer. In some cases (e.g., a major corporate or governmental system) acceptance testing can be very formal and encompass many days or even weeks of testing.

SYSTEM TESTING

At the beginning of this book, we stressed the fact that software is only one element of a larger computer-based system. Ultimately, software is incorporated with other system elements (e.g., hardware, people, information), and a series of system integration and validation tests are conducted. These tests fall outside the scope of the software process and are not conducted solely by software engineers. However, steps taken during software design and testing can greatly improve the probability of successful software integration in the larger system.

A classic system-testing problem is “finger pointing.” This occurs when an error is uncovered, and the developers of different system elements blame each other for the problem. Rather than indulging in such nonsense, you should anticipate potential interfacing problems and (1) design error-handling paths that test all information coming from other elements of the system, (2) conduct a series of tests that simulate bad data or other potential errors at the software interface, (3) record the results of tests to use as “evidence” if finger pointing does occur, and (4) participate
in planning and design of system tests to ensure that software is adequately tested.

i) Recovery Testing

Many computer-based systems must recover from faults and resume processing with little or no downtime. In some cases, a system must be fault tolerant; that is, processing faults must not cause overall system function to cease. In other cases, a system failure must be corrected within a specified period of time or severe economic damage will occur. *Recovery testing* is a system test that forces the software to fail in a variety of ways and verifies that recovery is properly performed. If recovery is automatic (performed by the system itself), reinitialization, check pointing mechanisms, data recovery, and restart are evaluated for correctness. If recovery requires human intervention, the mean-time-to-repair (MTTR) is evaluated to determine whether it is within acceptable limits.

ii) Security Testing

Any computer-based system that manages sensitive information or causes actions that can improperly harm (or benefit) individuals is a target for improper or illegal penetration. Penetration spans a broad range of activities: hackers who attempt to penetrate systems for sport, disgruntled employees who attempt to penetrate for revenge, dishonest individuals who attempt to penetrate for illicit personal gain. *Security testing* attempts to verify that protection mechanisms built into a system will, in fact, protect it from improper penetration. “The system’s security must, of course, be tested for invulnerability from frontal attack—but must also be tested for invulnerability from flank or rear attack.” Given enough time and resources, good security testing will ultimately penetrate a system. The role of the system designer is to make penetration cost more

WHITE-BOX TESTING

White-box testing, called **glass-box testing** is a test case design method that uses the control structure of the procedural design to derive test cases.

Using white-box testing methods, the software engineer can derive test cases that

1. guarantee that all independent paths within a module have been exercised at least once,
2. exercise all logical decisions on their true and false sides,
3. execute all loops at their boundaries and within their operational bounds, and
4. exercise internal data structures to ensure their validity.

"Why spend time and energy worrying about (and testing) logical minutiae when we might better expend effort ensuring that program requirements have been met?" or “Why don't we spend all of our energy on black-box tests?”

The answer is :

Logic errors and incorrect assumptions are inversely proportional to the probability that a program path will be executed. Errors tend to creep into our work when we design and implement function, conditions, or controls that are out of the mainstream. Everyday processing tends to be well understood (and well scrutinized), while "special case" processing tends to fall into the cracks.

We often believe that a logical path is not likely to be executed when, in fact, it may be executed on a regular basis. The logical flow of a program is sometimes counterintuitive, meaning that our unconscious assumptions about flow of control and data may lead us to make design errors that are uncovered only once path testing commences.

Typographical errors are random. When a program is translated into programming language source code, it is likely that some typing errors will occur. Many will be uncovered by syntax and type checking mechanisms, but others may go undetected until testing begins. It is as likely that a typo will exist on an obscure logical path as on a mainstream path.

Each of these reasons provides an argument for conducting white-box tests. Black-box testing, no matter how thorough, may miss the kinds of errors noted here. White-box testing is far more likely to uncover them.

BASIS PATH TESTING

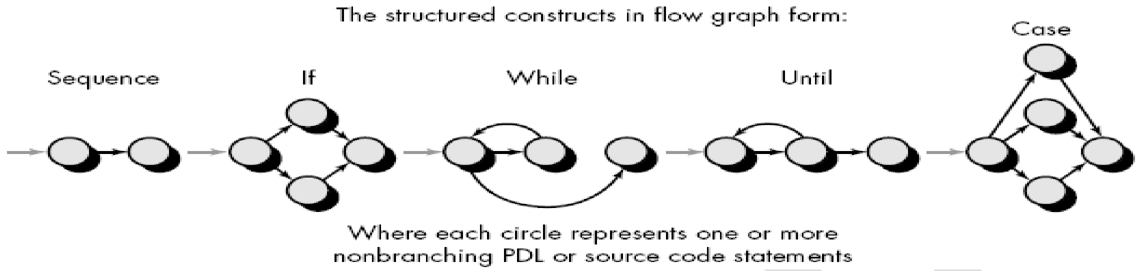
Basis path testing is a white-box testing technique first proposed by **Tom McCabe** in 1976.

The basis path method enables the test case designer to derive a logical complexity measure of a procedural design and use this measure as a guide for defining a basis set of execution paths.

Test cases derived to exercise the basis set are guaranteed to execute every statement in the program at least one time during testing.

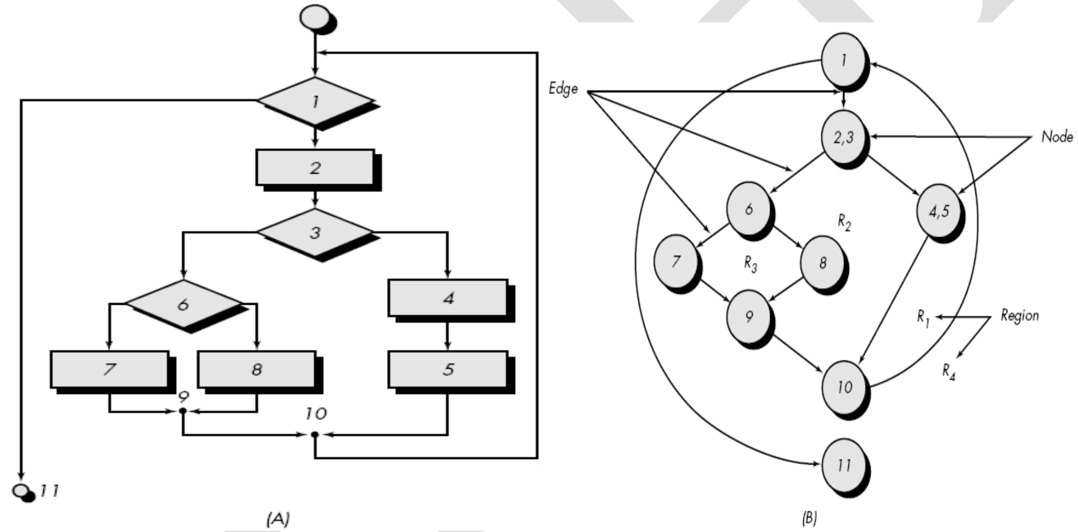
Flow Graph Notation

The flow graph depicts logical control flow using the notation illustrated in Fig 5.1.


The structured constructs in flow graph form:
Sequence
If
While
Until
Case
Where each circle represents one or more nonbranching PDL or source code statements

Flow graph notation

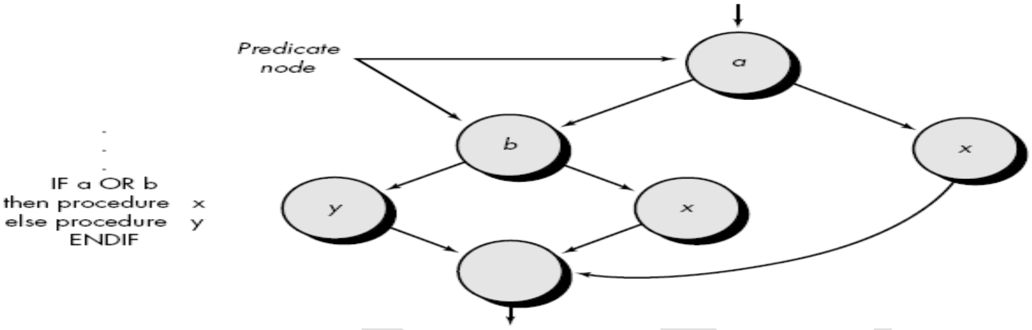
Each structured construct has a corresponding flow graph symbol. To illustrate the use of a flow graph, we consider the procedural design representation in Fig 5.2A. Here, a flowchart is used to depict program control structure.



Flowchart, (A) and flow graph (B)

- Fig maps the flowchart into a corresponding flow graph (assuming that no compound conditions are contained in the decision diamonds of the flowchart).
- Referring to Fig, each circle, called a flow graph node, represents one or more procedural statements.
- A sequence of process boxes and a decision diamond can map into a single node.
- The arrows on the flow graph, called edges or links, represent flow of control and are analogous to flowchart arrows.

- An edge must terminate at a node, even if the node does not represent any procedural statements (e.g., see the symbol for the if-then-else construct).
- Areas bounded by edges and nodes are called regions. When counting regions, we include the area outside the graph as a region.4
- When compound conditions are encountered in a procedural design, the generation of a flow graph becomes slightly more complicated.
- A compound condition occurs when one or more Boolean operators (logical OR, AND, NAND, NOR) is present in a conditional statement.
- Referring to Fig 5.3, the PDL segment translates into the flow graph shown.
- **Note:** A separate node is created for each of the conditions a and b in the statement IF a OR b. Each node that contains a condition is called a predicate node and is characterized by two or more edges emanating from it.



```
.
.
.
IF a OR b
then procedure   x
else procedure   y
ENDIF
```

Fig 5.3 Compound logic

Cyclomatic Complexity

Cyclomatic complexity is **software metric that provides a quantitative measure of the logical complexity of a program.**

Cyclomatic complexity has a foundation in graph theory and provides us with extremely useful software metric.

Cyclomatic complexity is defined by the number of independent paths in the basis set of a program and provides us with an upper bound for the number of tests that must be conducted to ensure that all statements have been executed at least once.

An independent path is any path through the program that introduces at least one new set of processing statements or a new condition. When stated in terms of a flow graph, an independent path must move along at least one edge that has not been traversed before the path is defined.

For example, a set of independent paths for the flow graph illustrated in Fig 5.2B is

path 1: 1-11

path 2: 1-2-3-4-5-10-1-11

path 3: 1-2-3-6-8-9-10-1-11

path 4: 1-2-3-6-7-9-10-1-11

Note:Each new path introduces a new edge.

The path **1-2-3-4-5-10-1-2-3-6-8-9-10-1-11** is not considered to be an independent path because it is simply a combination of already specified paths and does not traverse any new edges.

Paths 1, 2, 3, and 4 constitute a basis set for the flow graph in Fig 5.2B. That is, if tests can be designed to force execution of these paths (a basis set), every statement in the program will have been guaranteed to be executed at least one time and every condition will have been executed on its true and false sides.

Note: The basis set is not unique. In fact, a number of different basis sets can be derived for a given procedural design.

How do we know how many paths to look for? The computation of cyclomatic complexity provides the answer.

Complexity is computed in one of three ways:

1. The number of regions of the flow graph corresponds to the cyclomatic complexity.
2. Cyclomatic complexity, V(G), for a flow graph, G, is defined as V(G) = E - N + 2 where E is the number of flow graph edges, N is the number of flow graph nodes.
3. Cyclomatic complexity, V(G), for a flow graph, G, is also defined as V(G) = P +1 where P is the number of predicate nodes contained in the flow graph G.

The Cyclomatic complexity of the flow graph in Fig 5.2B, can be computed using each of the algorithms just noted:

1. The flow graph has four regions.

2. V(G) = 11 edges - 9 nodes + 2 = 4.

3. V(G) = 3 predicate nodes + 1 = 4.

Therefore, the cyclomatic complexity of the flow graph in Figure 17.2B is 4.

Important: the value for V(G) provides us with an upper bound for the number of independent paths that form the basis set and, by implication, an upper bound on the number of tests that must be designed and executed to guarantee coverage of all program statements.

CONTROL STRUCTURE TESTING

The basis path testing technique is one of a number of techniques for control structure testing.

Other variations on control structure testing are discussed. These broaden testing coverage and improve quality of white-box testing.

Condition Testing

Condition testing is a test case design method that exercises the logical conditions contained in a program module.

A simple condition is a Boolean variable or a relational expression, possibly preceded with one NOT (¬) operator.

A **relational expression** takes the form **E1 <relational-operator> E2** where E1 and E2 are arithmetic expressions and <relational-operator> is one of the following: <, ≤, =, ≠ (nonequality), >, or ≥.

A **compound condition** is composed of two or more simple conditions, Boolean operators, and parentheses. We assume that Boolean operators allowed in a compound condition include OR (|), AND (&) and NOT (¬).

A condition without relational expressions is referred to as a **Boolean expression**. Therefore, the possible types of elements in a condition include a Boolean operator, a Boolean variable, a pair of Boolean parentheses (surrounding a simple or compound condition), a relational operator, or an arithmetic expression.

If a condition is incorrect, then at least one component of the condition is incorrect. Therefore, types of errors in a condition include the following:

- Boolean operator error (incorrect/missing/extra Boolean operators).
- Boolean variable error.
- Boolean parenthesis error.
- Relational operator error.
- Arithmetic expression error.

The condition testing method focuses on testing each condition in the program.

Condition testing strategies have two advantages.

1. Measurement of test coverage of a condition is simple.
2. Test coverage of conditions in a program provides guidance for the generation of additional tests for the program.

The purpose of condition testing is to detect not only errors in the conditions of a program but also other errors in the program.

A number of condition testing strategies have been proposed.

Branch testing is probably the simplest condition testing strategy. For a compound condition C, the true and false branches of C and every simple condition in C need to be executed at least once.

Domain testing requires three or four tests to be derived for a relational expression. For a relational expression of the form **E1 <relational-operator> E2** three tests are required to make the value of E1 greater than, equal to, or less than that of E2. If <relational-operator> is incorrect and E1 and E2 are correct, then these three tests guarantee the detection of the relational operator error. To detect errors in E1 and E2, a test that makes the value of E1 greater or less than that of E2 should make the difference between these two values as small as possible.

Data Flow Testing

The data flow testing method selects test paths of a program according to the locations of definitions and uses of variables in the program.

To illustrate the data flow testing approach, assume that each statement in a program is assigned a unique statement number and that each function does not modify its parameters or global variables.

For a statement with S as its statement number,

DEF(S) = {X | statement S contains a definition of X}

USE(S) = {X | statement S contains a use of X}

If statement S is an if or loop statement, its DEF set is empty and its USE set is based on the condition of statement S. The definition of variable X at statement S is said to be live at statement S' if there exists a path from statement S to statement S' that contains no other definition of X.

A definition-use (DU) chain of variable X is of the form [X, S, S'], where S and S' are statement numbers, X is in DEF(S) and USE(S'), and the definition of X in statement S is live at statement S'.

Data flow testing strategies are useful for selecting test paths of a program containing nested if and loop statements. To illustrate this, consider the application of DU testing to select test paths for the PDL that follows:

```
proc x
        B1;
        do while C1
        if C2
        then
        if C4
        then B4;
        else B5;
        endif;
        else
        if C3
        then B2;
        else B3;
        endif;
        endif;
        enddo;
        B6;
end proc;
```

To apply the DU testing strategy to select test paths of the control flow diagram, we need to know the definitions and uses of variables in each condition or block in the PDL.

Assume that variable X is defined in the last statement of blocks B1, B2, B3, B4, and B5 and is used in the first statement of blocks B2, B3, B4, B5, and B6. The DU testing strategy requires an execution of the shortest path from each of Bi, $0 < i \leq 5$, to each of Bj, $1 < j \leq 6$. Although there are 25 DU chains of variable X, we need only five paths to cover these DU chains. The reason is that five paths are needed to cover the DU chain of X from Bi, $0 < i \leq 5$, to B6 and other DU chains can be covered by making these five paths contain iterations of the loop.

Since the statements in a program are related to each other according to the definitions and uses of variables, the data flow testing approach is effective for error detection.

However, the problems of measuring test coverage and selecting test paths for data flow testing are more difficult than the corresponding problems for condition testing.

Loop Testing

Loops are the cornerstone for the vast majority of all algorithms implemented in software.

Loop testing is a white-box testing technique that focuses exclusively on the validity of loop constructs.

Four different classes of loops can be defined: simple loops, concatenated loops, nested loops, and unstructured loops (Fig).

Simple loops.

The following set of tests can be applied to simple loops, where n is the maximum number of allowable passes through the loop.

1. Skip the loop entirely.

2. Only one pass through the loop.

3. Two passes through the loop.

4. m passes through the loop where m < n.

5. n -1, n, n + 1 passes through the loop.

Nested loops.

If we were to extend the test approach for simple loops to nested loops, the number of possible tests would grow geometrically as the level of nesting increases.

Beizer suggests an approach that will help to reduce the number of tests:

1. Start at the innermost loop. Set all other loops to minimum values.

2. Conduct simple loop tests for the innermost loop while holding the outer loops at their minimum iteration parameter (e.g., loop counter) values. Add other tests for out-of-range or excluded values.

3. Work outward, conducting tests for the next loop, but keeping all other outer loops at minimum values and other nested loops to "typical" values.

4. Continue until all loops have been tested.

Concatenated loops.

Concatenated loops can be tested using the approach defined for simple loops, if each of the loops is independent of the other. However, if two loops are concatenated and the loop counter for loop 1 is used as the initial value for loop 2, then the loops are not independent. When the loops are not independent, the approach applied to nested loops is recommended.

Unstructured loops.

Whenever possible, this class of loops should be redesigned to reflect the use of the structured programming constructs.

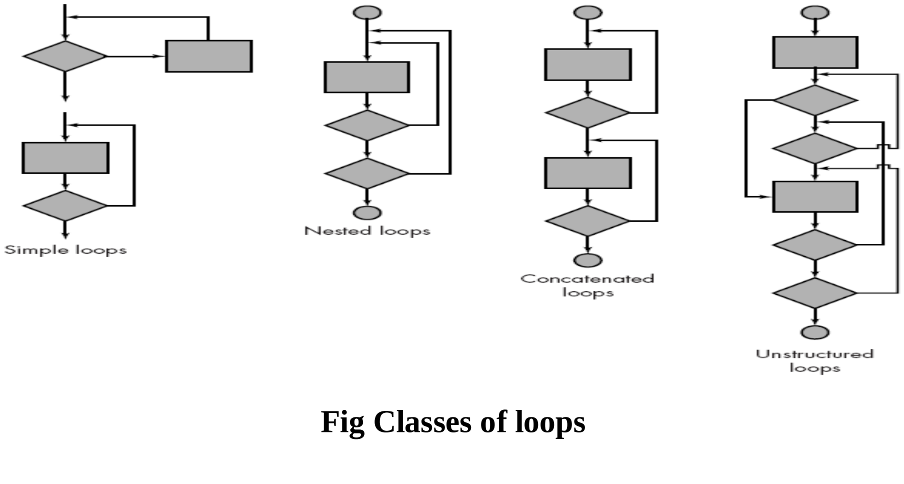


Fig Classes of loops

BLACK-BOX TESTING

Black-box testing, also called behavioral testing, focuses on the functional requirements of the software.

That is, black-box testing enables the software engineer to derive sets of input conditions that will fully exercise all functional requirements for a program. Black-box testing is not an alternative to white-box techniques.

Rather, it is a complementary approach that is likely to uncover a different class of errors than white-box methods.

Black-box testing attempts to find errors in the following categories:

1. Incorrect or missing functions
2. Interface errors
3. Errors in data structures or external data base access,
4. Behavior or performance errors, and
5. Initialization and termination errors.

Unlike white-box testing, which is performed early in the testing process, black-box testing tends to be applied during later stages of testing.

Black-box testing purposely disregards control structure, attention is focused on the information domain.

Tests are designed to answer the following questions:

• How is functional validity tested?

• How is system behavior and performance tested?

• What classes of input will make good test cases?

• Is the system particularly sensitive to certain input values?

• How are the boundaries of a data class isolated?

• What data rates and data volume can the system tolerate?

• What effect will specific combinations of data have on system operation?

Black-box techniques, we derive a set of test cases that satisfy the following criteria:

1. test cases that reduce, by a count that is greater than one, the number of additional test cases that must be designed to achieve reasonable testing and
2. test cases that tell us something about the presence or absence of classes of errors, rather than an error associated only with the specific test at hand.

Graph-Based Testing Methods

The first step in black-box testing is to understand the objects that are modeled in software and the relationships that connect these objects.

Next step is to define a series of tests that verify “all objects have the expected relationship to one another”.

To accomplish these steps, the software engineer begins by creating a **graph**—a collection of nodes that represent objects; links that represent the relationships between objects; node weights that describe the properties of a node (e.g., a specific data value or state behavior); and link weights that describe some characteristic of a link.

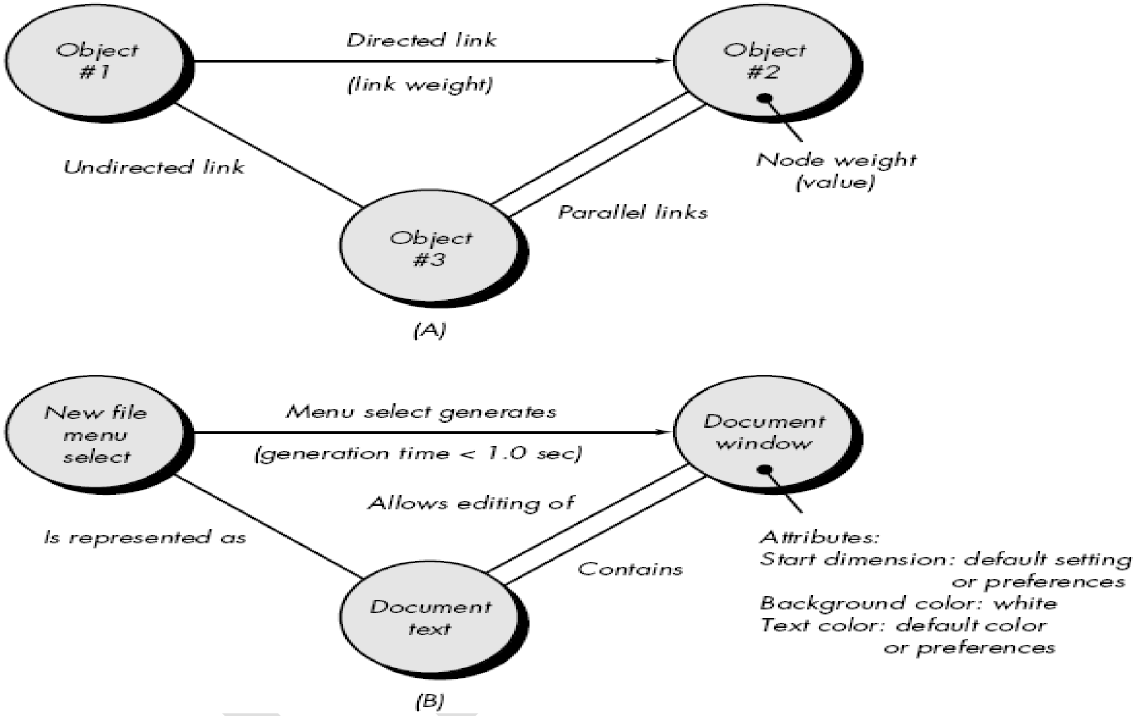


Fig (A) Graph notation (B) Simple example

The symbolic representation of a graph is shown in Fig A.

Nodes are represented as circles connected by links that take a number of different forms. A directed link (represented by an arrow) indicates that a relationship moves in only one direction.

A bidirectional link, called a **symmetric link**, implies that the relationship applies in both directions. Parallel links are used when a number of different relationships are established between graph nodes.

Eg. consider a portion of a graph for a word-processing application (Fig B) where

Object #1 = new file menu select

Object #2 = document window

Object #3 = document text

Referring to the figure, a menu select on new file generates a document window.

The node weight of document window provides a list of the window attributes that are to be expected when the window is generated.

The link weight indicates that the window must be generated in less than 1.0 second.

An undirected link establishes a symmetric relationship between the new file menu select and document text, and parallel links indicate relationships between document window and document text.

In reality, a far more detailed graph would have to be generated as a precursor to test case design.

The software engineer then derives test cases by traversing the graph and covering each of the relationships shown. These test cases are designed in an attempt to find errors in any of the relationships.

Beizer describes a number of behavioral testing methods that can make use of graphs:

Transaction flow modeling. The nodes represent steps in some transaction (e.g., the steps required to make an airline reservation using an on-line service), and the links represent the logical connection between steps (e.g., flight. information. input is followed by validation/availability. processing).

Finite state modeling. The nodes represent different user observable states of the software (e.g., each of the “screens” that appear as an order entry clerk takes a phone order), and the links represent the transitions that occur to move from state to state (e.g., order-information is verified during inventory-availability look-up and is followed by customer-billing-information input). The state transition diagram can be used to assist in creating graphs of this type.

Data flow modeling. The nodes are data objects and the links are the transformations that occur to translate one data object into another. For example, the node FICA.tax.withheld (FTW) is computed from gross.wages (GW) using the relationship, FTW = 0.62 - GW.

Timing modeling. The nodes are program objects and the links are the sequential connections between those objects. Link weights are used to specify the required execution times as the program executes.

Graph-based testing begins with the definition of all nodes and node weights. That is, objects and attributes are identified. The data model can be used as a starting point, but it is important to note that many nodes may be program objects (not explicitly represented in the data model). To provide an indication of the start and stop points for the graph, it is useful to define entry and exit nodes.

Once nodes have been identified, links and link weights should be established.

In general, links should be named, although links that represent control flow between program objects need not be named.

Each relationship is studied separately so that test cases can be derived.

The transitivity of sequential relationships is studied to determine how the impact of relationships propagates across objects defined in a graph. Transitivity can be illustrated by considering three objects, X, Y, and Z. Consider the following relationships:

X is required to compute Y

Y is required to compute Z

Therefore, a transitive relationship has been established between X and Z:

X is required to compute Z

Based on this transitive relationship, tests to find errors in the calculation of Z must consider a variety of values for both X and Y.

The symmetry of a relationship (graph link) is also an important guide to the design of test cases.

As test case design begins, the first objective is to achieve node coverage. By this we mean that tests should be designed to demonstrate that no nodes have been inadvertently omitted and that node weights (object attributes) are correct.

Next, link coverage is addressed. Each relationship is tested based on its properties. For example, a symmetric relationship is tested to demonstrate that it is, in fact, bidirectional. A transitive relationship is tested to demonstrate that transitivity is present.

A reflexive relationship is tested to ensure that a null loop is present. When link weights have been specified, tests are devised to demonstrate that these weights are valid. Finally, loop testing is invoked

Equivalence Partitioning

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

An ideal test case single-handedly uncovers a class of errors (e.g., incorrect processing of all character data) that might otherwise require many cases to be executed before the general error is observed.

Equivalence partitioning strives to define a test case that uncovers classes of errors, thereby reducing the total number of test cases that must be developed.

Test case design for equivalence partitioning is based on an evaluation of equivalence classes for an input condition.

An equivalence class represents a set of valid or invalid states for input conditions. Typically, an input condition is either a specific numeric value, a range of values, a set of related values, or a Boolean condition.

Equivalence classes may be defined according to the following guidelines:

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

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

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

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

Example, consider data maintained as part of an automated banking application.

The user can access the bank using a personal computer, provide a six-digit password, and follow with a series of typed commands that trigger various banking functions. During the log-on sequence, the software supplied for the banking application accepts data in the form

area code—blank or three-digit number

prefix—three-digit number not beginning with 0 or 1

suffix—four-digit number

password—six digit alphanumeric string

commands—check, deposit, bill pay, and the like

The input conditions associated with each data element for the banking application can be specified as

area code: Input condition, Boolean—the area code may or may not be present.

Input condition, range—values defined between 200 and 999, with specific exceptions.

prefix: Input condition, range—specified value >200

Input condition, value—four-digit length

password: Input condition, Boolean—a password may or may not be present.

Input condition, value—six-character string.

command: Input condition, set—containing commands noted previously.

Applying the guidelines for the derivation of equivalence classes, test cases for each input domain data item can be developed and executed. Test cases are selected so that the largest number of attributes of an equivalence class are exercised at once.

Boundary Value Analysis

Boundary value analysis leads to a selection of test cases that exercise bounding values.

Boundary value analysis is a test case design technique that complements equivalence partitioning.

Rather than selecting any element of an equivalence class, BVA leads to the selection of test cases at the "edges" of the class. Rather than focusing solely on input conditions, BVA derives test cases from the output domain as well.

Guidelines for BVA are similar in many respects to those provided for equivalence partitioning:

1. If an input condition specifies a range bounded by values a and b, test cases should be designed with values a and b and just above and just below a and b.
2. If an input condition specifies a number of values, test cases should be developed that exercise the minimum and maximum numbers.
3. Apply guidelines 1 and 2 to output conditions.
4. If internal program data structures have prescribed boundaries (e.g., an array has a defined limit of 100 entries), be certain to design a test case to exercise the data structure at its boundary.

By applying these guidelines, boundary testing will be more complete, thereby having a higher likelihood for error detection.

Comparison Testing

There are some situations (e.g., aircraft avionics, automobile braking systems) in which the reliability of software is absolutely critical.

In such applications redundant hardware and software are often used to minimize the possibility of error.

When redundant software is developed, separate software engineering teams develop independent versions of an application using the same specification.

In such situations, each version can be tested with the same test data to ensure that all provide identical output. Then all versions are executed in parallel with real-time comparison of results to ensure consistency.

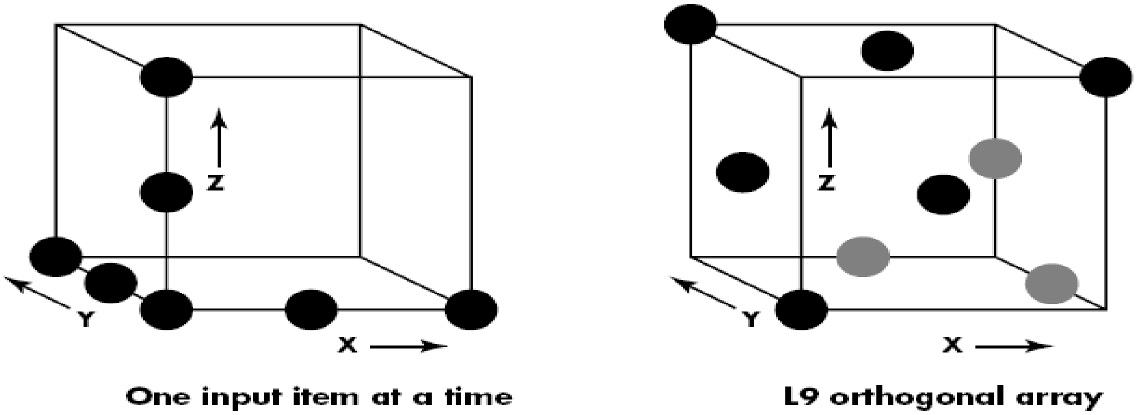
Researchers have suggested that independent versions of software be developed for critical applications, even when only a single version will be used in the delivered computer-based system.

These independent versions form the basis of a black-box testing technique called **comparison testing** or **back-to-back testing**.

When multiple implementations of the same specification have been produced, test cases designed using other black-box techniques (e.g., equivalence partitioning) are provided as input to each version of the software.

If the output from each version is the same, it is assumed that all implementations are correct. If the output is different, each of the applications is investigated to determine if a defect in one or more versions is responsible for the difference. In most cases, the comparison of outputs can be performed by an automated tool.

Comparison testing is not foolproof. If the specification from which all versions have been developed is in error, all versions will likely reflect the error. In addition, if each of the independent versions produces identical but incorrect results, condition testing will fail to detect the error.



A geometric view of test cases

Orthogonal Array Testing

There are many applications in which the input domain is relatively limited. That is, the number of input parameters is small and the values that each of the parameters may take are clearly bounded. When these numbers are very small, it is possible to consider every input permutation and exhaustively test processing of the input domain.

However, as the number of input values grows and the number of discrete values for each data item increases, exhaustive testing become impractical or impossible.

Orthogonal array testing can be applied to problems in which the input domain is relatively small but too large to accommodate exhaustive testing.

The orthogonal array testing method is particularly useful in finding errors associated with region faults—an error category associated with faulty logic within a software component.

PART A(Online)

PART B (2 Marks)

1. Define abstraction.
2. What do you mean by an error?
3. Differentiate between refinement and refactoring
4. Compare black box and white box testing
5. Write the difference between transform flow and transaction flow
6. List the different types of loops in testing
7. What is transform mapping?
8. What is validation testing?
9. Define transaction mapping.
10. What is the use of system testing?

PART C (6 Marks)

1. Explain Graph based testing methods in Black Box testing.
2. Demonstrate Flow graph notation and Independent program path in Basis path testing.
3. Demonstrate in detail about Validation testing
4. Explain in detail about Equivalence Partitioning
5. Discuss about Boundary value analysis.
6. Write in detail about Software Testing Fundamentals.
7.Illustrate in detail about System testing.
8. Write short notes on condition testing.
9. Illustrate the use of dataflow testing in software engineering process.
10. Discuss in detail about orthogonal array testing.
11. Illustrate loop testing and its types.

KARPAGAM ACADEMY OF HIGHER EDUCATION

DEPARTMENT OF COMPUTER SCIENCE

II BSc CS BATCH(2016 - 2019)

PART-A OBJECTIVE TYPE/ MULTIPLE CHOICE QUESTIONS

UNIT-5

	Questions	Opt1	Opt2	Opt3	Opt4	Answer
SNO	Validation focuses on ______________.	the ability of the interface to implement every user task correctly	the degree to which the interface is easy to use and easy to learn.	the user’s acceptance of the interface as a useful tool in their work.	all of the above.	all of the above.
1	__________ is a critical element of software quality assurance and represents the ultimate review of specification, design, and code generation.	software specification	software generation	software coding	software testing	software testing
2	Software is tested from ___________ different perspectives.	2	3	4	5	2
3	Software engineers are by their nature ___________ people.	pessimistic	optimistic	constructive	destructive	constructive
4	__________ is a process of executing a program with the intent of finding an error.	coding	testing	debugging	designing	testing
5	All tests should be _________ to customer requirements.	traceable	designed	tested	coded	traceable
6	Tests should be planned long before _____________ begins.	testing	coding	specification	requirements	testing
7	Testing should begin in the _________ and progress toward testing in the large.	design	beginning	small	big	small
8	The less there is to test, the more _________ we can test it.	quickly	shortly	automatically	hardly	quickly
9	________ is a process of executing a program with the intend of finding an error.	testing	coding	planning	designing	testing
10	A good _________ is one that has a high probability of finding an as-yet-undiscovered error	planning	test case	objective	goal	test case
11	All _________ should be traceable to customer-requirements.	analysis	designs	tests	plans	tests
12	__________ is simple how easily a computer program can be tested.	software operability	software simplicity	software decomposability	software testability	software testability
13	The better it works, the more efficiently it can be testing. This characteristic is called ___________.	operability	observability	controllability	decomposability	operability
14	There are _________ characteristics in testability	5	6	7	8	7
15	What you see is what you test. This characteristic is called __________.	controllability	observability	decomposability	stability	observability
16	The better we can control the software, the more the testing can be automated and optimized. This characteristic is called	operability	stability	understandability	controllability	controllability
17	By controlling the scope of testing, we can more quickly isolate problems and perform smarter retesting. This characteristic is called	decomposability	simplicity	stability	understandability	decomposability
18	. The less there is to test, the more quickly we can test it. This characteristic is called _________.	controllability	simplicity	operability	observability	simplicity
19	The fewer the changes, the fewer the disruptions to testing. This characteristic is called __________.	controllability	decomposability	stability	understandability	stability
20	. The more information we have, the smarter we will test. This characteristic is called _________.	controllability	decomposability	stability	understandability	understandability
21	A good test has a high ___________ of finding an error.	probability	simplicity	understandability	stability	probability
22	A good test is not _________.	stable	redundant	simple	complex	redundant
23	White-box testing sometimes called _________.	control structure testing	condition testing	glass-box testing	black-box testing	glass-box testing
24	Logic errors and incorrect assumptions are inversely proportional to the ___________ that a program path will be executed	simplicity	probability	understandability	stability	probability
25	Typographical errors are _________.	redundant	simple	random	complex	random
26	One often believes that a _________ path is not likely to be executed when, in fact, it may be executed on a regular basis	control	structural	physical	logical	logical
27	Basic path testing is a __________.	black-box testing	white-box testing	control structure testing	control path testing	white-box testing
28	__________ is a software metric that provides a quantitative measure of the logical complexity of a program	cyclomatic complexity	flow graph	deriving test cases	graph matrices	cyclomatic complexity
29	An __________ is any path through the program that introduces atleast one new set of processing statements or a new condition	dependent path	independent path	basic path	control path	independent path
30	There are _________ steps to be applied to derive the basis set.	2	3	4	5	4

31	There are _________ test cases that satisfy the basis set.	3	4	5	6	6
32	. A ________ is a square matrix whose size is equal to the number of nodes on the flow graph.	graph matrix	matrix	flow graph	cyclomatic complexity	graph matrix
33	To develop a software tool that assists in basis path testing, a data structure called a ___________ is useful	matrix	flow graph	graph matrix	cyclomatic omplexity	graph matrix
34	____________ requires three or four tests to be derived for a relational expression.	branch testing	data flow testing	data control testing	domain testing	domain testing
35	__________ is probably the simplest condition testing strategy.	branch testing	data flow testing	condition testing	domain testing	branch testing
36	The __________ method selects test paths of a program according to the locations of definitions and uses of variables in the program	data flow testing	condition testing	loop testing	black box testing	data flow testing
37	__________ is a white box testing technique that focuses exclusively on the validity of loop constructions	data flow testing	loop testing	condition testing	control path testing	loop testing
38	___________ is a test case design method that exercises the logical conditions contained in a program module	black box testing	loop testing	data flow testing	condition testing	condition testing
39	_____________ is called behavioral testing.	black box testing	loop testing	data flow testing	condition testing	black box testing
40	The first step in __________ is to understand the objects that are modeled in software and the relationships that connect these objects	black box testing	loop testing	data flow testing	condition testing	black box testing
41	Equivalence partitioning is a ___________ method that divides the input domain of a program into classes of data	black box testing	loop testing	data flow testing	condition testing	black box testing
42	Comparison testing is also called ____________.	black box testing	loop testing	behavioral testing	back-to-back testing	back-to-back testing
43	__________ testing can be applied to problems in which the input domain is relatively small but too large to accommodate exhaustive testing	orthogonal array	loop	behavioral	back-to-back	orthogonal array
44	__________ focuses verification effort on the smallest unit of software design – the software component or module	module testing	unit testing	structure testing	system testing	unit testing
45	A driver is nothing more than a __________.	subprogram	main program	stub	subroutine	main program
46	_____________ serve to replace modules that are subordinate called by the component to be tested.	subprograms	main programs	stubs	subroutines	stubs
47	Drivers and _________ represent overhead.	subprograms	main programs	stubs	subroutines	stubs
48	___________ of execution paths is an essential task during the unit test.	unit testing	module testing	selective testing	white box testing	selective testing
49	Good _________ dictates that error conditions be anticipated and error-handling paths set up to reroute or cleanly terminate processing when an error does	design	testing	code	module	design
50	_________ is completely assembled as a package, interfacing errors have been uncovered and corrected.	software	program	code	all of the above	software
51	Thread testing is used for testing	Real time systems	Object oriented systems	Event driven systems	All of the above	Object oriented systems
52	Testing of software with actual data and in the actual environment is called	Alpha testing	Beta testing	Regression testing	None of the above	Beta testing
53	Functionality of software is tested by	White box testing	Black box testing	Regression testing	None of the above	Black box testing
54	Integration testing techniques are	Top down	Bottom up	Sandwich	All of the above	All of the above
55	Testing the software is basically	Verification	Validation	Verification and validation	None of the above	Verification and validation
56	Which one is not the verification activity	Reviews	Path testing	Walkthrough	Acceptance testing	Acceptance testing
57	Alpha and Beta testing techniques are related to	System testing	Unit testing	Acceptance testing	Integration testing	Acceptance testing
58	A break in the working of a system is called	Defect	Failure	Fault	Error	Failure
59	Which is not a debugging techniques	Core dumps	Traces	Print statements	Regression testing	Regression testing
60	Data flow testing is related to	Data flow diagrams	E-R diagrams	Data dictionaries	none of the above	none of the above