20/9/23

Software Engineering (SE)

Course code

AGCS07

UNIT-1

Part-I

Introduction to Software Engineering

1. The evolving role of software (S/w)
2. Changing Nature of S/w
3. Software Myths

Part-II

A Generic view of Process

1. S/w Engineering - A layered Technology
2. A process Frame work
3. The capability Maturity Model Integration (CMMI)

Part-III

Process Models

1. The Waterfall Model
2. Spiral Model
3. Agile Methodology

Software Engineering

Introduction

The term software engineering is the product of two words, software & engineering.

- The software is a collection of integrated programs.
- Slw subseeds (maintain / support) of carefully organised instructions and code written by developers on any of various particular computer languages.
- Computer programs and related documentation such as requirements, design models and user manuals.
- Engineering is the application of scientific and practical knowledge to invent, design, build, maintain and improve frame works, processes etc.

Software Engineering

S.E is an engineering branch related to the evolution of software product using well-defined scientific principles, techniques and procedures.

The result of software engineering is an effective and reliable s/w product.

Goal

Goal of the SE is "to produce high quality software at low cost."

① The Evolving role of a software

S/w takes dual role. It is both a product and a vehicle (Process) for delivering a product.

As a product : It delivers the computing potential include by computer hardware (or) by a network of computers.

As a vehicle : It is information transformer- producing, managing, acquiring, modifying, displaying (or) transmitting information that can be a simple as single bit (or) as complex as a multimedia presentation.

Software delivers the most important product of our time - information.

→ It transforms personal data

→ It manages business information to enhance competitiveness:-

→ It provides a gate way to world-wide information networks.

→ It provides the means for acquiring information

The role of computer slw has undergone significant change over a span of little more than 50 years.

Dramatic improvements in hardware performance, profound changes in computer architectures, vast increases in memory and storage capacity and a wide-variety of exhautic input & output options have more sophisticated and complex computer based system.

The lone programmer of an earlier era has been replaced by a team of slw specialists, each focusing on one part of the technology required to deliver a complex application. And yet, the same questions asked by the lone programmer are being asked when modern ~~and~~ computer based systems are built:

→ Why does it takes so long to get slw finished?

→ Why are development costs so high?

→ Why can't we find all the errors before

we give the s/w to customers?

→ Why do we continue to have difficulty in measuring progress as s/w is being developed?

These and many other questions are a manifestation of the concern about s/w and the manner in which it is developed.

Today, a huge s/w industry has become a dominant factor in the economies of the industrialised world.

(2) Changing nature of Software

The nature of slw has changed a lot over the years. Seven broad categories of computer slw continuing challenges for software engineers:

1. System slw
2. Application slw
3. Engineer / Scientific slw
4. Embedded slw
5. Product-line slw
6. Web Applications
7. Artificial Intelligence slw

1. System Software

System software is a collection of programs written to service other programs. The system slw is the interface b/w the hardware and user applications. The operating system is the best known example of system slw.

Eg: Microsoft windows, compliers, editors

2. Application Software

Application software consist of standalone programs that solve a specific business need.

- Applications in this area process business (or) technical data in a way that facilitates business operations (or) management (or) technical decision making.

- In addition to conventional data processing, application slw is used to control business functions in real-time.

  Eg: Point-of-sale transaction processing, Realtime manufacturing process-control.

3. Embedded Software

3. Engineering / scientific slw.

Engineering / scientific slw satisfies the needs of a scientific (or) engineering user to perform enterprise-specific tasks. Such software is written for specific applications using principles, techniques & formulae particular to that field.

- However, modern applications within the engineering/scientific area are moving away from conventional numerical algorithms.

  Eg: MAT LAB, AUTOCAD, PSPICE, ORCAD etc..

4. Embedded slw

Embedded slw resides within a product (or) system and is used to implement and control features and functions for the end user and for the system itself.

- It can perform
  - * Limited and esoteric functions (eg: keyboard control for microwave ovens)
  - * Provide significant function and control capability eg: Digital functions in automobile such as fuel control, dash board displays, breaking

systems etc.,

5. Product-line software

Product-line s/w is designed to provide a specific capability for use by many different customers.

. Product-line s/w can

* focus on a limited and esoteric market place

Eg: Inventory control products

(or)

* Address mass consumer markets

Eg: Word processing, spreadsheets, computer graphics, multimedia, entertainment, database management, personal and business financial applications.

6. Web Applications

The software related to web come under this category.

Eg: CGI, HTML, JAVA, PERL, DHTML etc.,

. Web Applications are evolving into sophisticated computing environments that not only provide standalone features, computing functions and content to the end user, but also are integrated with corporate database and business applications.

7. Artificial Intelligence software

Artificial Intelligence slw makes use of non-numerical algorithms to solve complex problems that are not amenable to computation (or) straight forward analysis.

- Application within this area includes robotics, expert systems, pattern recognition (image & voice), artificial neural networks, theorem proving and game playing.

26/9/23

③ Software Myths

Slw myths - beliefs about slw and the process used to build it - can be traced to the earliest days of computing. Myths have a number of attributes that have made them insidious.

Most knowledgable professionals recognise myths for what they are misleading attributes that have caused serious problems for managers and technical people alike. However, old attitudes and habits are difficult to modify and remnants of slw myths are still believed.

i) Management Myths

Myth 1 : We already have a book i.e., full of standards and procedures for building s/w, won't that provide many people with everything they need to know ?

Reality : The book of standards may very well exist, but is it used ? Are s/w practitioners aware of its existence ? Does it reflect modern s/w engineer practice ? Is it complete ? Is it streamline to improve time and to delivery while still maintaining a focus on quality ? In many cases the answer to all of these questions is 'NO'.

Myth 2 : If we get behind schedule, we can add more programmers and catch up.

Reality : S/w development is not a mechanistic process like manufacturing.

"Adding people to a late software project makes it later". At first, the statement may seem counter-intuitive. However, as new people are added, people who were working must spend time educating the new comers, thereby reducing the amount of time spent on productive development effort. People can be added but only in a planned and well coordinated manner.

Myth 3: If I decide to outsource the slw project to a third-party, I can just relax and let that firm build it.

Reality: If an organization does not understand how to manage and control slw projects internally it will invariably struggle when it outsources slw projects.

ii) Customer Myths

Myth 1: A general statement of objectives is sufficient to begin writting programs - we can fill in the details later.

Reality: A poor upfront definition is the major cause of failed slw efforts. A formal and detailed descriptions of the information domain, function, behaviour, performance, interfaces, design constraints and validation criteria is essential. These characteristics can be determined only after thorough communication between customer and developer.

Myth 2: Project requirements continually change, but change can be easily accomidated because slw is flexible.

Reality: It is true that slw requirements change, but the impact of change varies with the time at which it is introduced. When

requirement changes are requested early (before design or code has been started) cost impact is relatively small. However, as time passes, the cost impact grows rapidly. Resources have been committed and a design framework has been established and change can cause upheaval that requires additional resources and major design modifications.

iii) Practitioner's Myths

Myth 1: Once we write the program and get it work, our job is done.

Reality: Someone once said that "The sooner you began 'writing code', the longer it will take you to get done". Industry data indicate that between 60 & 80 percent of all effort expended on slw will be expended after it is deliever to the customer for the first time.

Myth 2: Until I get the program running I have no way of assessing its quality.

Reality: One of the most effective slw quality assurance mechanisms can be applied ; from the inception of a project - the formal technical review (FTR). Slw reviews are a "Quality filter" that have been found to be more effective than testing for finding certain classes of slw errors.

Myth 3: The only deliverable work product for a successful project is the working program.

Reality: A working program is only one part of a s/w configuration, that includes many elements. Documentation provides a foundation for successful engineering and more important guidance for s/w support.

29/9/23

Part - II

Generic view of processing

① Software Engineering - A layered Technology

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

Bauers definition provides it with a baseline:

→ What "sound engineering principles" can be applied to computer s/w development?

→ How do we "economically" build software so that is "reliable"?

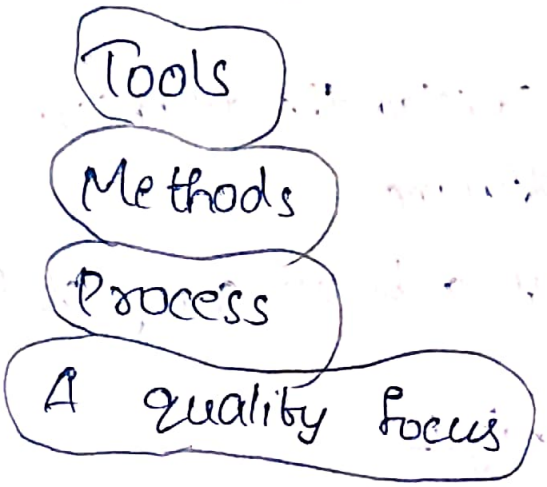
→ What is required to create computer programs that work "efficiently" on not one but many different "real machines"?

These are the qsns that continue to challenge s/w engineers

The IEEE developed a more comprehensive definition when it states <u>software engineering</u>.

* The application of a systematic, discipline, quantifiable approach to the development, operation & maintainence of slw. That is the application of engineering to slw.
* The study of approaches as in the above definition,

Software engineering is a layered technology



1. Quality Focus

Any engineering approach must rest on an organisational commitment to quality. The bedrock that supports S.E is a quality focus.

2. Process

The foundation for S.E is the process layered S.E process is the glue that holds the technology layers together and enables rational & timely development of computer slw. Process defines a framework that must be establish for effective delivery of slw engineering technology

The s/w process forms the basis for management control of s/w projects & establishes the context in which

* Technical methods are applied
* Work products are produced
* Milestones are established
* Quality is ensured and
* Change is properly managed.

3. Methods

S.E methods provide the technical "how to's" for building software

Methods encompass a broad array of tasks that include

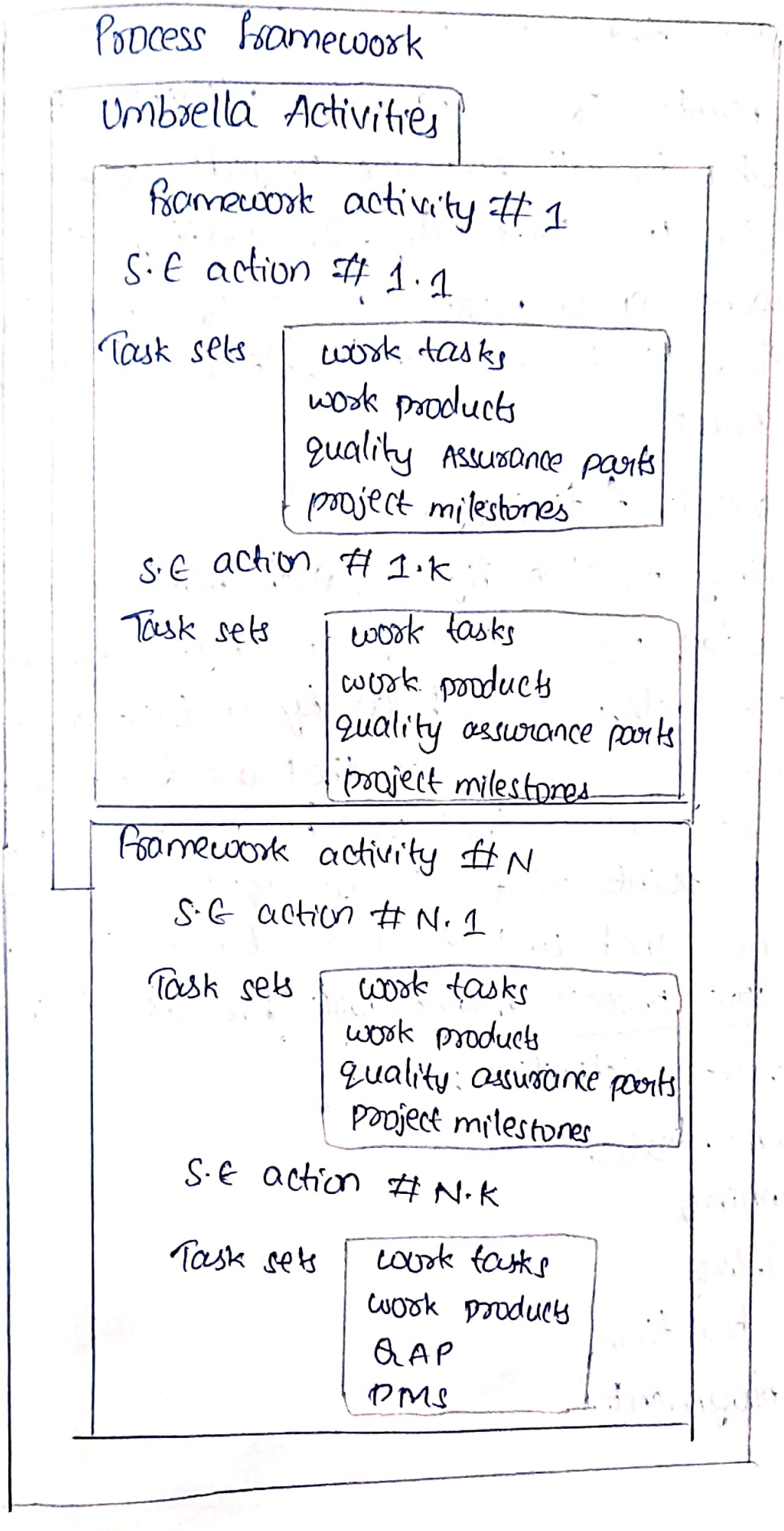
* Communication
* Requirements analysis
* Design Modeling
* Program construction
* Testing & support

4. Tools

S.E 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 tool, a system for the support of s/w development called computer aided s/w engineering is established.

(2) Process framework



4/10/23

A process framework establishes the foundation for a complete s/w process by identifying a small number of framework activities that are applicable to all software projects, regardless of their size (or) complexity. In addition, the process framework encompasses a set of umbrella activities that are applicable across the entire s/w process

Framework Activities:

Referring to the figure, each framework activity is included by a set of 's/w engineering actions. Each S.E action is defined by a task set that identifies the work tasks that are to be completed, the work products that will be produced, the quality assurance points that will be required, and the milestones that will be used to indicate progress

A generic process framework for S.E defines 5 framework activities:

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

1. Communication :

This framework activity involves heavy communication & collaboration with the customer and stake holders and encompasses requirements gathering and other related activities.

2. Planning:

This activity establishes a plan for the s/w engineering work that follows. It describes the technical task to be conducted, the risks that are likely, the resources that will be required, the work products to be produced and a workschedule.

3. Modeling;

This activity encompasses the creation of models that allow the developer and customer to better understand s/w requirements and the design that will achieve those requirements.

The modeling activity is composed of two S.E actions

i) Analysis
ii) Design

i) Analysis : It encompasses a set of work tasks (Eg: requirements gathering, elaboration, negotiation, specification & validation) that lead to creation of analysis model.

ii) Design : It encompasses work tasks (data design, architectural design, interface design & component-level design) that create a design model.

4. Construction:

This activity code generation and the testing that is required to uncover the errors in the code.

5. Deployment:

The slw is delivered to the customer who evaluates the delivered product and provides feedback based on the evaluation.

These 5 generic framework activities can be used during the development of small programs, the creation of large web applications and for the engineering of large, complex computer based systems. The details of the slw process will be quite different in each case, but the framework activities remain the same.

5/10/23

Umbrella Activities:

The following are the set of umbrella activities

1. Software project tracking & control: It allows the slw team to asses progress against the project plan and take necessary action to maintain schedule

2. Risk Management: Asses risk that may effect the outcome of the project (or) the quality of the product.

3. Software Quality Assurance : It defines and conducts the activities required to ensure slw quality and also it performs actions to ensure the product's quality. This maintain time complexity, space complexity & overall quality of that product.

4. Formal Technical Reviews (FTR) : It asses S.E work products in an effort to uncover and remove errors before they are propagated to the next action (or) activity.

5. Measurement : It define & collects and process, project and product measures that assist the team in delivering slw that needs customer's needs can be used in conjunction with all other framework and umbrella activities.

6. Slw Configuration Management : It manages the effects of change throughout the slw process. Managing of configuration process is, when any change in the slw occurs. SCM has some rules & techniques because there are different linking library files, linking slw files. So it maintain the configuration of all these.

7. Reusability Management: It defines criteria for work product reuse and establishes mechanisms to achieve reusable components. Reusable work items should be backed up.

Eg: Login modules, Logout modules

8. Work product preparation & production: It encompasses the activities required to create work products such as models, documents, logs, forms & lists. It maintain the documentation of each & every activity.

(3) CMMI (Capability Maturity Model Integration)

- CMMI is a successor of CMM
- It is a more evolved model that incorporates best components of individual disciplines of CMM like slw CMM, systems engineer CMM, people CMM etc
- Since CMM is a reference model of matured practices in a specific displines, so it becomes difficult to integrate these disciplines as per the requirements.

Objectives of CMMI:

1. Fulfilling customer needs and expectations.
2. Value creation for investors / stock holders.
3. Market growth is increased.
4. Improved quality of products and services.

5. Enhanced reputation in Industry.

CMMI Representation:

A representation allows an organization to pursue a different set of improvement objectives. There are two representations for CMMI.

1. Staged Representation
2. Continuous Representation

1. Staged Representation:

- Uses a pre-defined set of process areas to define improvement path.
- Provides a sequence of improvements, where each part in the sequence serves as a foundation for the next.
- An improved path is defined by maturity level.
- Maturity level describes the maturity of processes in organization.
- Staged CMMI representation allows comparison b/w different organisations for multiple maturity levels.

2. Continuous Representation:

- Allows selection of specific process area.
- Uses capability levels that measures improvement of individual process area.
- Continuous CMMI Representation allows comparison b/w different organizations on a process-area

by process-area basis.

- Allows organisations to select processes which require more improvement.
- In this representation, order of improvement of various processes can be selected which allows the organisations to meet their objectives and eliminate risks.

CMMI Model - Maturity levels:

In CMMI with staged representation, there are five maturity levels.

i) Maturity level 1 : Initial

→ Processes are poorly managed or controlled.
→ Unpredictable outcomes of processes involved.
→ Adhoc and Chaotic approach used.
→ No KPA (Key Process Area) defined.
→ Lowest quality and highest risk.

ii) Maturity Level 2 : Managed

→ Requirements are managed.
→ Processes are planned and controlled.
→ Projects are managed and implemented according to their documented plans.
→ Risk involved in this level is lower than initial level, but still exists.
→ Quality is better than initial level.

iii) Maturity level 3: Defined

→ Processes are well characterised and described using standards proper procedures and methods, tools, etc...

→ Medium quality and medium risk involved.

→ Focus is process standardization.

iv) Maturity Level 4: Quantitatively Managed

→ Quantitative objects for process performance and quality are set.

→ Quantitative objectives are based on customer requirements, organization needs, etc.

→ Process performance measures are analysed quantitatively.

→ Higher quality of processes is achieved.

→ Lower risk.

v) Maturity Level 5: Optimizing

→ Continuous improvement in processes and their performance.

→ Improvement has to be both incremental and innovative.

→ Highest quality of processes.

→ Lowest risk in processes and their performance.

CMMI Model - capability Levels:

A capability level includes relevant specific and generic practices for a specific process area that can improve the organization's processes associated with that process area.

→ For CMMI models with continuous representation, there are six capability levels.

ⓐ Capability level 0: Incomplete

* Incomplete process - partially or not performed.
* One or more specific goals of process area are not met.
* No generic goals are specified for this level.
* This capability level is same as maturity level 1.

ⓑ Capability level 1: Performed

* Process performance may not be stable.
* Objectives of quality, cost & schedule may not be met.
* A capability level 1 process is expected to perform all specific and generic practices for this level.
* Only a start-step for process improvement.

ⓒ Capability level 2: Managed

* Process is planned, monitored and controlled.
* Managing the process by ensuring that objectives are achieved.

* Objectives are both model and other including cost, quality, schedule.
* Actively managing processing with the help of metrics.

(d) Capability level 3: Defined
* A defined process is managed and meets the organization's set of guidelines and standards.
* Focus is process standardization.

(e) Capability level 4: Quantitatively Managed
* Process is controlled using statistical & quantitative techniques.
* Process performance and quality is understood in statistical terms and metrics.
* Quantitative objectives for process quality and performance are established.

(f) Capability level 5: Optimizing
* Focus on continually improving process performance.
* Performance is improved in both ways - incremental and innovation.
* Emphasizes on studying the performance results across the organisation to ensure that common causes or issues are identified and fixed.

12/10/23 (11) Process Models

Introduction:-

In S.E, a slw process model is the mechanism of building slw development work into distinct phases to improve design, product, management & project management it is also known as slw development life cycle.

It establishes the foundation for a complete slw process by identifying a small no. of framework activities. It includes a set of umbrella activities that are applicable across the entire slw process. Each framework activity is populated by a set of S.E actions. A generic process framework for S.E encompasses five activities.

(i) Communication
(ii) Planning
(iii) Modeling
(iv) Construction
(v) Deployment

→ Process modeling is the graphical representation of business process (or) work flows. Slw process is the set of activities & associated outcome that produce a slw product, which are common to all slw processes.

Waterfall Model :- (WFM)

The WFM, sometimes called the classic lifecycle, suggests a systematic sequential approach to slw development that begins with customer specification of requirements & progress through planning, modeling, construction & deployment.

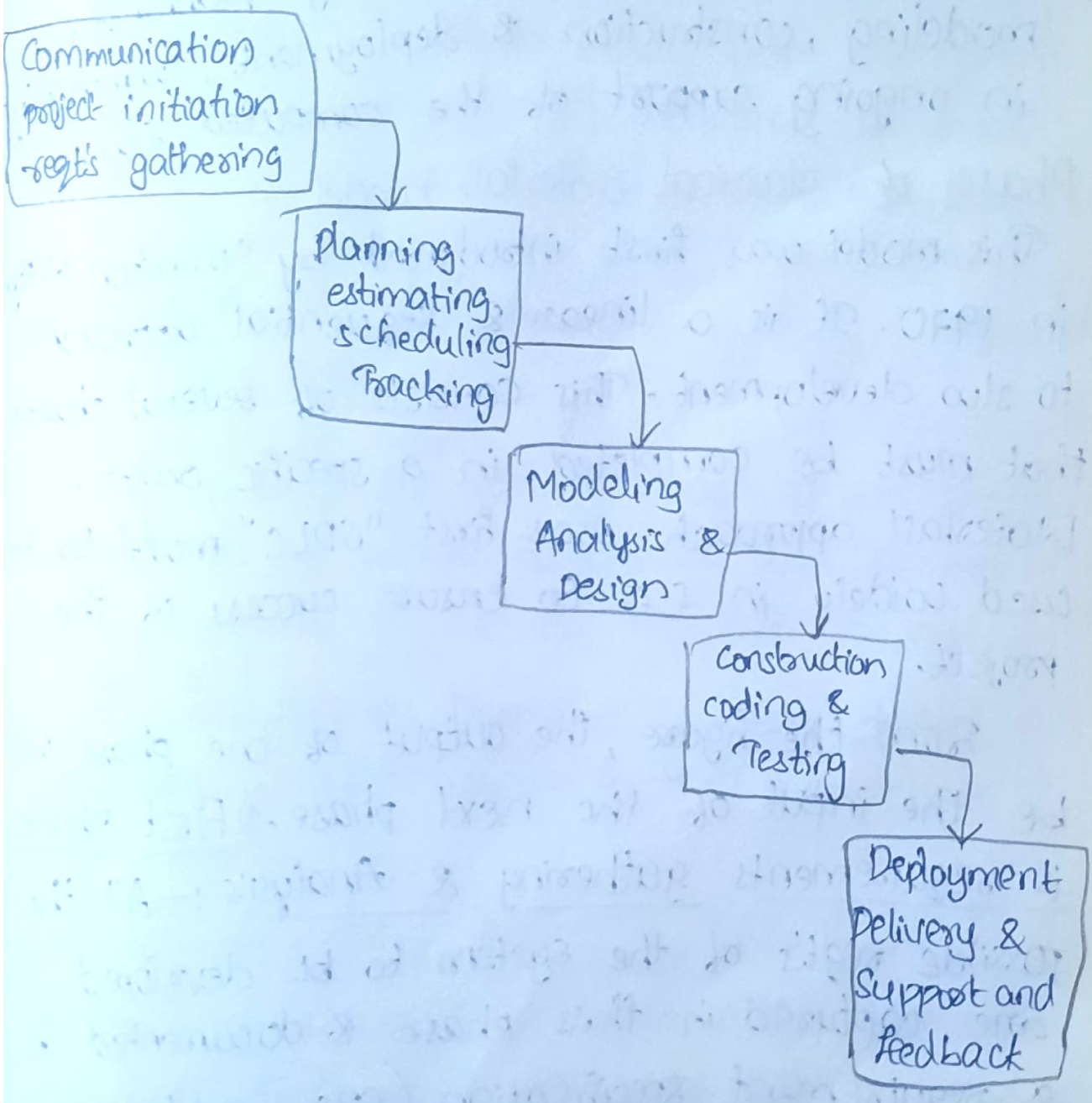


fig: The waterfall Model.

The Waterfall Model (WFM):-

It is often used for large scale projects with long timelines, where there is a little room (scope) for errors & project stakeholders need to have high level of confidence in the outcome. This model begins with customer specification of requirements & progressive through planning, modeling, construction & deployment, culminating in ongoing support of the computed s/w.

Phases of classical waterfall model:-

This model was first introduced by "winston W. Royce" in 1970. It is a linear & sequential approach to s/w development. This consists of several phases that must be completed in a specific order. Waterfall approach was first "SDLC" model to be used widely in S.E to ensure success of the project.

From the figure, the output of one phase will be the input of the next phase. 1) Phase-1 First phase is requirements gathering & Analysis:- All the possible reqts of the system to be developed are captured in this phase & documented in a requirement specification (SRS - Software Requirements Specifications).

phase-2
2) System Design :- The requirements specifications from 1st phase are studied & the system design is prepared. This system design helps in specifying hardware & system requirements & helps in defining the overall system architecture.

phase-3
3) Implementation :- With inputs from the system design, the system is 1st developed in small programs called units, which are integrated in the next phase. Each unit is developed & tested for its functionality, which is referred to as "unit testing".

phase-4
4) Testing :- Once the testing phase comes, the s/w is tested as a whole to ensure that it meets the requirements & it is free from defects.

phase-5
5) Deployment :- Once the s/w has been tested & approved, it is deployed to the production environment.

phase-6
6) Maintenance :- The final phase which involves fixing any issues that arise after the s/w has been deployed & ensuring that it continues to meet the requirements over the time.

Advantages of WFM :-

Easy to understand : It is very simple and easy to understand.

Individual processing : Phases in this model are processed one at a time.

Properly defined : Each stage in this model is clearly defined.

Clear milestone : It has very clear and well-understood milestones.

Properly documented : Process, actions & results are well documented.

Reinforces good habits : This model reinforces good habits like define before design and design before code.

Working :

It works well for smaller projects & project where requirements are well understood.

This waterfall model has several benefits as it helps projects keep a well defined predictable project under budget.

Disadvantages of WFM :-

Because of some major drawbacks of this model. We can't use it in real projects, but we use other software development lifecycle

models which are based on the classical waterfall model.

No feedback path : This model assumes that no error is ever committed by developers during any phase. Therefore it doesn't incorporate any mechanism for error correction.

Difficult to accommodate change requests : When customer requirements keep on changing with time, it is difficult to accommodate any change requests after the requirements specification phase is complete.

No overlapping of phases : In this model new phase can start only after the completion of previous phase. But in real project this can't be maintained. To increase efficiency & reduce cost phases may overlap.

Limited flexibility : Which is not well-suited for projects with changing or uncertain requirements. Once a phase has been completed it is difficult to make changes or go back to a previous phase.

Limited stake-holder involvement : The stakeholders are typically involved in the early phases of the project requirements gathering and analysis but may not be involved in the later

phases (implementation, testing, deployment)

Lengthy development cycle: This model can result in lengthy development cycle as each phase may be completed before moving on to the next. This can result in delays & increased costs if requirement changes or new issues arise.

Not suitable for complex projects: This model can make it difficult to manage multiple dependencies and inter-related components.

Applications of WFM :-

Large scale slw development projects: This model ensures that, this project is completed on time and within budget.

Projects with well-defined Requirements: As the sequential nature of the model requires a clear understanding of the projects objectives & scope

Projects with stable requirements: This model is well-suited for projects with stable requirements as the linear nature of the model does not allow for changes to be made once a phase has been completed.

22/11/23

(2) Spiral Model

It comes under iterative process model it is an evaluationary model. It has 4 stages. It is an endless loop, it never ends, it contains repeatative activities in this, where risks are predictable then we can use spiral model. It starts at Q_1 and ends at Q_4. In this the customer evaluation will be done. If the customer is not satisfied then one more iteration will go up. Like that again this spiral with a loop continues until & unless the customer is satisfied.

In first iteration it contains the core product and in next iteration it contains the subsequent phases and at last the product will be completed. The first phase is one round i.e., 360°.

The radius of the spiral increases, cost also increases as the radius keeps on increasing, the cost also increases. Everything will increase as the radius of the spiral keeps on increasing.

Angle will indicate the progress. For example, the angle is 180° - half of the task is done. The angle will represent the progress of the task spiral is looped until customer is satisfied. This is suitable for large projects. And also also is flexible, complex & it takes time.

Define objective:

Requirements gathering & analysis.

Identify & resolve risks:

In this identify all the risks and try to resolve those risks.

Developing next version of slw:

Nothing but interacting with the customers and taking the feedback from them, if there are any changes then those changes will be implemented.

Review & plan for next phase:

Next phase is nothing but the second iteration all those decisions will be made in this step.

If 1 module is completed then deliver it to the customer and it can be released to the customer when that module is stable, then only we take it to the next module, we test it then only we perform the integration between the 2nd module & 1st module. Then after we test everything we can release it to customer. Spiral model is also called as iterative model and also called incremental model.

Advantages:

- Requirement changes are allowed after every cycle.
- It is also called as controlled model.
- Testing is done for every cycle, before going to the next cycle.

- Customer will get to use the slw for every module.
- It is a controlled model to adapt since here we test the module & once it is stable then only we can go for the next module.

Disadvantages

- Requirement changes are not allowed in between the cycle.

```
#include<stdio.h>
#include<stdlib.h>
struct Node{
        int data;
        struct Node* link;

};
struct Node* start = Null;
struct Node* createNode(int data){
        struct Node * newNode =(struct Node*)mallo
                                        (sizeof(struct
                                        Node
        if(!newNode){
          printf("memory allocation failed.\n");
          exit(1);
        }
        newNode->data = data;
        newNode-> link = NULL;
        return newNode;
    }
void insertFront(int data) {
      Struct Node* newNode =(createNode(data);
       newNode-> link = start;
       start-= NewNode;
}
void insertRear (int data) {
      struct Node* newNode = createNode(data).
             if(!start)start =NewNode;
```

```
else {
    struct Node* current = start;
    while (current->link) current = current->link;
    current->link = newNode;
  }
}
```

AGILE MODEL

Topics covered

1 About Agile Model

2. Traditional vs Agile model working with example

3. When to use the Agile model

4. Agile principles.

5. Advantages of Agile Model

6. Disadvantages of Agile Model.

About Agile Model

→ Mostly used model in today's digital era.

→ Agile means "The ability to respond to changes from requirements technology and people".

→ It is an incremental and iterative process of software development.

2. working with example:

→ Divides requirements into multiple iterations and provides specific functionality for the release.

→Delivers multiple software requirements

→Each iteration lasts for two to three weeks.

→Direct Collaboration with customers.

→ Rapid project development.

Traditional vs Agile Model working with Example

For example:

→ Instagram social Application:

Requirements are.

1. Follow-unfollow option

2. Edit profile

3. search

4. Messaging

5. Post photos

6. upload story

7. To make reels.

8. Go Live

→ Agile model divides the complete requirements into the multiple iterations.

→ And they develop the product as per the priority of the requirements

→ In the below paragraphs, the time required for development of the above example is taken both for Waterfall model and Agile model.

Suppose in Waterfall Model, it takes two months for requirements gathering and analysis phase. After that it take 1/1/2 months for designing pupose and after that it takes 4 months approximately for coding purpose and 1 1/2 month for testing pupose and at the end, if any changes are required by the customer, let us assume the time to be for one month. We require 8 to 9 months for development purpose only.

and

Now in Agile Model in first iteration they take 3 to 4 weeks for development. After developing first iteration, they move to second iteration for 3 to 4 week and again move to next 3 to 4 weeks. For development purpose they require only 3 months. This is the reason why Agile model is in use in each and every software Development now a days. Agile model requires minimum time for development with greater accuracy and greater quality of the product.

WHEN TO USE THE AGILE MODEL

1. When project's size is large
2. When frequent changes are required. A pro
3. When highly qualified and experienced team is available
4. When a customer is ready to have a meeting with a software team all the time after each and every iteration.
5. Projects with flexible timelines and budget.

AGILE PRINCIPLES (12 PRINCIPLES)

1. Highest priority is to satisfy the customers to early and continue delivery of software
2. Being flexible about changing requirements at any point of development.
3. Working on frequent and short deliveries like couple of weeks or months with preference
4. Transparency between business people and developers and requires them to work together.
5. By providing a better productive environment and providing them with all the support, motivation. It leads to better productivity.
6. Face to face communication as the most effective way to communicate between customer and development team.
7. Continuous attention towards effective designing and Technical Excellence through following optimal code standard.
8. It promotes sustainable development because of the work of developers, users and sponsors as all of them work together.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity is the art of maximum result and less hard work by removing unnecessary tasks and prioritizing activities.
11. The best architectures, requirements and designs emerge from self-organizing and experience teams.
12. For developing effective software, regular analysis and work on improving the overall delivery or the development process.

Advantages:

1. Supports customer involvement and customer satisfaction.
2. Strong Communication of the software team with the customer.
3. Little planning required.
4. Efficient design and fulfils the business requirement.
5. Anytime changes are acceptable.
6. Provides a very realistic approach to software development.
7. Updated versions of functioning software are released every week.
8. It reduces total development time

Disadvantages:

1. Due to lack of proper documentation, once the project completes and the developers are allotted to another project, maintenance of the finished project can become difficult.
2. Depends heavily on customer interaction, so if the customer is not clear, team can be driven in the wrong direction.

UNIT - II

SOFTWARE REQUIREMENTS & REQUIREMENTS ENGINEERING PROCESS

Part I

Software Requirements:

1. Functional and non-functional requirements
2. user requirements
3. system requirements
4. interface specification
5. the software requirements document

Part II

Requirementsengineering process:

1. Feasibility studies
2. Requirements elicitation and analysis
3. Requirements validation
4. Requirements management

Part I

Part I

Software Requirements:

1. Functional and non-functional requirements
2. user requirements
3. system requirements
4. interface specification
5. the software requirements document

1. Functional and non-functional requirements

Functional and Non-functional Requirements:

Software system requirements are often classified as functional requirements ornonfunctional requirements:

1. ***Functional requirements:*** These are statements of services the system should provide, how the system should react to particular inputs, and how the system should behave in particular situations.

In some cases, the functional requirements may also explicitly state what the system should not do.

2. ***Non-functional requirements****:* These are constraints on the services or functions offered by the system.

They include timing constraints, constraints on the development process, and constraints imposed by standards.

Non-functional requirements often apply to the system as a whole, ratherthan individual system features or services.

Functional requirements:

The functional requirements for a system describe what the system should do. These requirements depend on

- the type of software being developed,
- the expected users of the software, and
- The general approach taken by the organization when writing requirements.

When expressed as user requirements, functional requirements are usually described in an abstract way that can be understood by system users. However, more specific functional system requirements describe the system functions, its inputs and outputs, exceptions, etc., in detail.

Functional system requirements vary from general requirements coveringwhat the system should do to very specific requirements reflecting local ways of working or an organization's existing systems.

Example:

Functional requirements for the MHC-PMS system, used to maintain information about patients receiving treatment for mental health problems:

1. A user shall be able to search the appointments lists for all clinics.
2. The system shall generate each day, for each clinic, a list of patients who are expected to attend appointments that day.
3. Each staff member using the system shall be uniquely identified by his or her eight-digit employee number.

In principle, the functional requirements specification of a system should be both complete and consistent.

- Completeness means that all services required by the user should be defined.
- Consistency means that requirements should not have contradictory definitions.

In practice, for large, complex systems, it is practically impossible to achieve requirements consistency and completeness.

One reason for this is that it is easy to make mistakes and omissions when writing specifications for complex systems.

Another reason is that there are many stakeholders in a large system. A stakeholder is a person or role that is affected by the system in some way. Stakeholders have different and often inconsistent needs. These inconsistencies may not be obvious when the requirements are first specified, so inconsistent requirements are included in the specification.

The problems may only emerge after deeper analysis or after the system has been delivered to the customer.

Non-functional requirements:

Non-functional requirements are the requirements that are not directly concerned with the specific services delivered by the system to the users.

They may relate to emerge system properties such as reliability, response time, and store occupancy. Alternatively, they may define constraints on the system implementation such as the capabilities of I/O devices or the data representations used in interfaces with other systems.

Non-functional requirements are often more critical than individual functional requirements. However, failing to meet a non-functional requirement can mean that the whole system is unusable. For example, if an aircraft system does not meet its reliability requirements, it will not be certified as safe for operation.

Although it is often possible to identify which system components implement specific functional requirements (e.g., there may be formatting components that implement reporting requirements), it is often more difficult to relate components to non-functional requirements. The implementation of these requirements may be diffused throughout the system. There are two reasons for this:

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

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

Classification of non-functional requirements:

Non-functional requirements arise through user needs, because of budget constraints, organizational policies, the need for interoperability with other softwareor hardware systems, or external factors such as safety regulations or privacy legislation.

Following figure shows the classification of non-functional requirements.

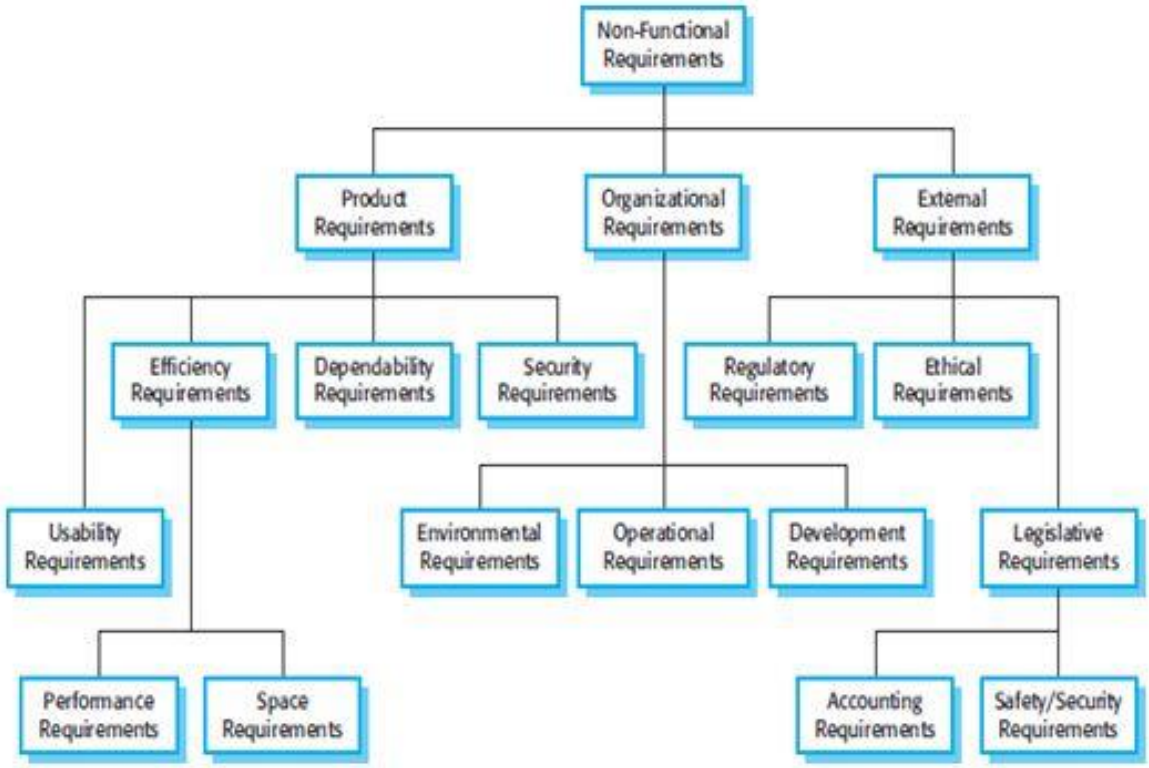


Figure: Types of Non-functional Requirements

From this diagram we can see that the non-functional requirements may come from required characteristics of the software (product requirements), the organization developing the software (organizational requirements), or from external sources

1. ***Product requirements:*** These requirements specify or constrain the behavior of the software.

Examples include performance requirements on how fast the system must execute and how much memory it requires, reliability requirements that set out the acceptable failure rate, security requirements, and usability requirements.

2. ***Organizational requirements:*** These requirements are broad system requirements derived from policies and procedures in the customer's and developer's organization.

Examples include operational process requirements that define how the system will be used, development process requirements that specify the programming language, the development environment or process standards to be used, and environmental requirements that specify the operating environment of the system.

3. ***External requirements:*** This broad heading covers all requirements that arederived from factors external to the system and its development process.

These may include regulatory requirements that set out what must be done for the system to be approved for use by a regulator, such as a central bank; legislative requirements that must be followed to ensure that the system operates within the law; and ethical requirements that ensure that the system will beacceptable to its users and the general public.

Example:

PRODUCT REQUIREMENT
The MHC-PMS shall be available to all dinics during normal working hours (Mon–Fri, 08.30–17.30). Downtime within normal working hours shall not exceed five seconds in any one day.

ORGANIZATIONAL REQUIREMENT
Users of the MHC-PMS system shall authenticate themselves using their health authority identity card.

EXTERNAL REQUIREMENT
The system shall implement patient privacy provisions as set out in HStan-03-2006-priv.

Whenever possible, we should write non-functional requirements quantitatively so that they can be objectively tested. Following figure shows metrics that you can use to specify non-functional system properties. We can measure these characteristics when the system is being tested to check whether or not thesystem has met its nonfunctional requirements.

Property	Measure
Speed	Processed transactions/second User/event response time Screen refresh time
Size	Mbytes Number of ROM chips
Ease of use	Training time Number of help frames
Reliability	Mean time to failure Probability of unavailability Rate of failure occurrence Availability
Robustness	Time to restart after failure Percentage of events causing failure Probability of data corruption on failure
Portability	Percentage of target dependent statements Number of target systems

Figure: Metrics for specifying non-functional requirements

Non-functional requirements such as reliability, safety, and confidentiality requirements are particularly important for critical systems.

Functional Requirements	Non-Functional Requirements
1. A functional requirement defines a system or its component.	A non-functional requirement defines the quality attribute of a software system.
2. It specifies "What should the software system do?"	It places constraints on "How should the software system fulfil the functional requirements?"
3. Functional requirement is specified by User.	Non-functional requirement is specified by technical peoples e.g. Architect, Technical leaders and software developers.
4. It is mandatory.	It is not mandatory.
5. It is captured in use case.	It is captured as a quality attribute.
6. Defined at a component level.	Applied to a system as a whole.
7. Helps you verify the functionality of the software.	Helps you to verify the performance of the software.
8. Functional Testing like System, Integration, End to End, API testing, etc are done.	Non-Functional Testing like Performance, Stress, Usability, Security testing, etc are done.
9. Usually easy to define.	Usually more difficult to define.
Example **1)** Authentication of user whenever he/she logs into the system. **2)** System shutdown in case of a cyber attack. **3)** A Verification email is sent to user whenever he/she registers for the first time on some software system.	**Example** **1)** Emails should be sent with a latency of no greater than 12 hours from such an activity. **2)** The processing of each request should be done within 10 seconds **3)** The site should load in 3 seconds when the number of simultaneous users are > 10000

User requirements:

These requirements describe what the end-user wants from the software system. User requirements are usually expressed in natural language and are typically gathered through interviews, surveys, or user feedback.

1. User requirements are written for customers
2. They are usually expresses in natural language.
3. Because of this, they are easy to understand
4. They describe services and features provided by system
5. This may include diagrams and tables which are understood by system users
6. The system users do not need technical knowledge to understand these
7. User requirements are for client managers, system end users, client engineers, contractor managers and system architects
8. They are gathered through various means such as interviews, surveys, or user feedback.

Example:

User Requirement Definition

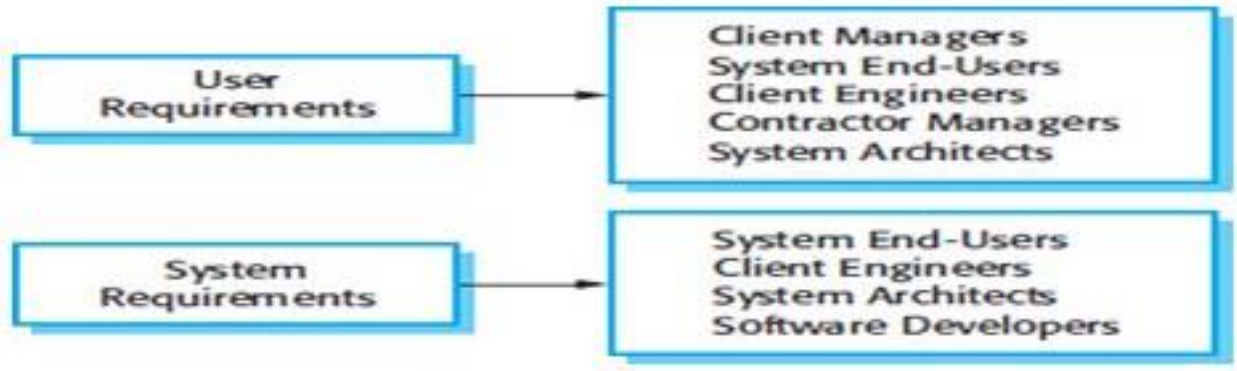
1. The MHC-PMS shall generate monthly management reports showing the cost of drugs prescribed by each clinic during that month.

System Requirements Specification

1.1 On the last working day of each month, a summary of the drugs prescribed, their cost, and the prescribing clinics shall be generated.
1.2 The system shall automatically generate the report for printing after 17.30 on the last working day of the month.
1.3 A report shall be created for each clinic and shall list the individual drug names, the total number of prescriptions, the number of doses prescribed, and the total cost of the prescribed drugs.
1.4 If drugs are available in different dose units (e.g., 10 mg, 20 mg) separate reports shall be created for each dose unit.
1.5 Access to all cost reports shall be restricted to authorized users listed on a management access control list.

This example from a mental health care patient management system (MHC- PMS) shows how a user requirement may be expanded into several system requirements.

Different levels of requirements are useful because they communicate information about the system to different types of reader.



System requirements:

These requirements specify the technical characteristics of the software system, such as its architecture, hardware requirements, software components, and interfaces. System requirements are typically expressed in technical terms and are often used as a basis for system design.

Salient features

1. Written for implementation team
2. They are written in technical language / technical terms
3. System Requirements describe the detailed description of services, features and complete operations of system
4. System Requirements may include system models and system designs
5. System Requirements can be understood by implementation team with technical knowledge.
6. System Requirements are for architects, software Developers, client engineers, system users and overall implementation team
7. They form basis for a system design

Interface Specification:

What is Interface?

- A point where two systems, subjects, organizations, etc. meet and interact.
- A device or program enabling a user to communicate with a computer.
- A inter face is a intersection between system and environment.
- Interface =system /environment

What is specification?

- A Specification is a agreement Between the produce of the services Consumer of that services

What is Interface specification?

- All software systems must operate with existing systems that have already been implemented and installed in an environment.
- If the new system and existing systems must work together, the interfaces of existing systems have to be precisely specified.
- These specifications should be defined early in the process and included in the requirements document.

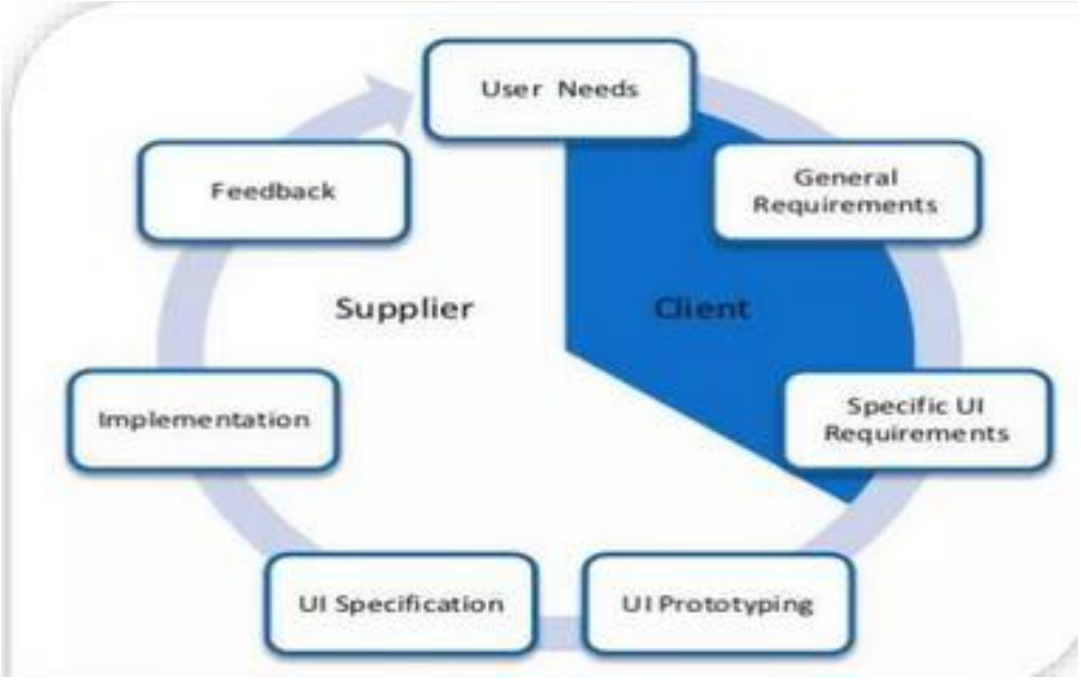
Types of Interface Specification:

- **Procedural interfaces.**
- **Data structures.**
- **Representations of data.**

Procedural interfaces where existing programs or sub-systems offer a range of services that are accessed by calling interface procedures. In simple words it Is used for calling the existing programs by the new programs These interfaces are sometimes called Application Programming Interfaces (APLs).

- **Data structures** that are passed from one sub-system to another. Graphical data models are the best notations for this type of description

- **Representations of data** (such as the ordering of bits) that have been established for an existing sub-system. These interfaces are most common in embedded, real-time system. Some programming languages such as Ada (although not Java) support this level of Specification.
- Sub system requesting service from other sub systems.



For 5 marks question

Interface Specification:

Most systems must operate with other systems and the operating interfaces mustbe specified as part of the requirements.

Three types of interface may have to be defined

- ***Procedural interfaces*** where existing programs or sub-systems offer a range of services that are accessed by calling interface procedures. These interfaces are sometimes called Application Programming Interfaces (APIs)
- ***Data structures that are exchanged*** that are passed from one sub-systemto another. Graphical data models are the best notations for this type of description
- ***Data representations*** that have been established for an existing sub-

system Formal notations are an effective technique for interface specification.

oOo

Software Requirements Document:

The software requirements document (sometimes called the software requirements specification or SRS) is an official statement of what the system developers should implement.

It should include both the user requirements for a system and a detailed specification of the system requirements.

Sometimes, the user and system requirements are integrated into a single description. In other cases, the user requirements are defined in an introduction to the system requirements specification. If there are a large number of requirements, the detailed system requirements may be presented in a separate document.

Requirements documents are essential when an outside contractor is developing the software system. The requirements document has a diverse set of users, ranging from the senior management of the organization that is paying for the system to the engineers responsible for developing the software. Following figure shows possible users of the document and how they use it.

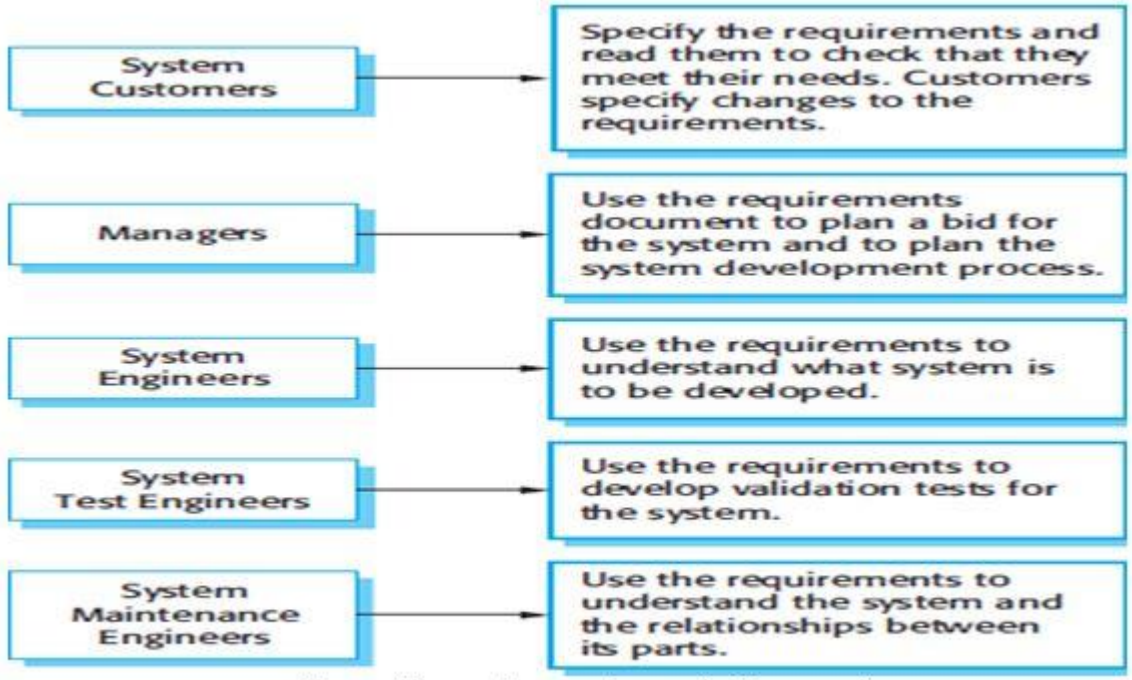


Figure: Users of a requirements document

The image in the next page shows one possible organization for a requirements document that is based on an IEEE standard for requirements documents (IEEE, 1998). This standard is a generic standard that can be adapted to specific uses.

Chapter	Description
Preface	This should define the expected readership of the document and describe its version history, including a rationale for the creation of a new version and a summary of the changes made in each version.
Introduction	This should describe the need for the system. It should briefly describe the system's functions and explain how it will work with other systems. It should also describe how the system fits into the overall business or strategic objectives of the organization commissioning the software.
Glossary	This should define the technical terms used in the document. You should not make assumptions about the experience or expertise of the reader.
User requirements definition	Here, you describe the services provided for the user. The non-functional system requirements should also be described in this section. This description may use natural language, diagrams, or other notations that are understandable to customers. Product and process standards that must be followed should be specified.
System architecture	This chapter should present a high-level overview of the anticipated system architecture, showing the distribution of functions across system modules. Architectural components that are reused should be highlighted.
System requirements specification	This should describe the functional and non-functional requirements in more detail. If necessary, further detail may also be added to the non-functional requirements. Interfaces to other systems may be defined.
System models	This might include graphical system models showing the relationships between the system components, the system, and its environment. Examples of possible models are object models, data-flow models, or semantic data models.
System evolution	This should describe the fundamental assumptions on which the system is based, and any anticipated changes due to hardware evolution, changing user needs, and so on. This section is useful for system designers as it may help them avoid design decisions that would constrain likely future changes to the system.
Appendices	These should provide detailed, specific information that is related to the application being developed; for example, hardware and database descriptions. Hardware requirements define the minimal and optimal configurations for the system. Database requirements define the logical organization of the data used by the system and the relationships between data.
Index	Several indexes to the document may be included. As well as a normal alphabetic index, there may be an index of diagrams, an index of functions, and so on.

In this case, they have extended the standard to include information about predicted system evolution. This information helps the maintainers of the system and allows designers to include support for future system features.

Requirements specification:

Requirements specification is the process of writing down the user and system requirements in a requirements document.

Ideally, the user and system requirements should be clear, unambiguous, easy to understand, complete, and consistent. In practice, this is difficult to achieve as stakeholders interpret the requirements in different ways and there are often inherent conflicts and inconsistencies in the requirements.

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

System requirements are expanded versions of the user requirements that are used by software engineers as the starting point for the system design. They add detail and explain how the user requirements should be provided by the system. They may be used as part of the contract for the implementation of the system and should therefore be a complete and detailed specification of the whole system. Ideally, the system requirements should simply describe the external behavior of the system and its operational constraints. They should not be concerned with howthe system should be designed or implemented.

User requirements are almost always written in natural language supplemented by appropriate diagrams and tables in the requirements document. System requirements may also be written in natural language but other notations based on forms, graphical system models, or mathematical system models can also be used. Following figure summarizes the possible notations that could be used for writing system requirements.

Notation	Description
Natural language sentences	The requirements are written using numbered sentences in natural language. Each sentence should express one requirement.
Structured natural language	The requirements are written in natural language on a standard form or template. Each field provides information about an aspect of the requirement.
Design description languages	This approach uses a language like a programming language, but with more abstract features to specify the requirements by defining an operational model of the system. This approach is now rarely used although it can be useful for interface specifications.
Graphical notations	Graphical models, supplemented by text annotations, are used to define the functional requirements for the system; UML use case and sequence diagrams are commonly used.
Mathematical specifications	These notations are based on mathematical concepts such as finite-state machines or sets. Although these unambiguous specifications can reduce the ambiguity in a requirements document, most customers don't understand a formal specification. They cannot check that it represents what they want and are reluctant to accept it as a system contract.

Ways of writing system requirements specification

oOo

Part II

Part II

Requirementsengineering process:

1. Feasibility studies
2. Requirements elicitation and analysis
3. Requirements validation
4. Requirements management

Requirements Engineering Process:

The requirements engineering process aims to produce an agreed requirements document that specifies a system satisfying stakeholder requirements.

Requirements are usually presented at two levels of detail. End-users and customers need a high-level statement of the requirements; system developers needa more detailed system specification.

There are four main activities in the requirements engineering process:

1. ***Feasibility study*** (These focus on assessing if the system is useful to thebusiness),
2. ***Requirements elicitation and analysis*** (discovering requirements),
3. ***Requirements specification*** (Converting these requirements i nto some standard form), and
4. ***Requirements validation*** (checking that the requirements actually define thesystem that the customer wants).

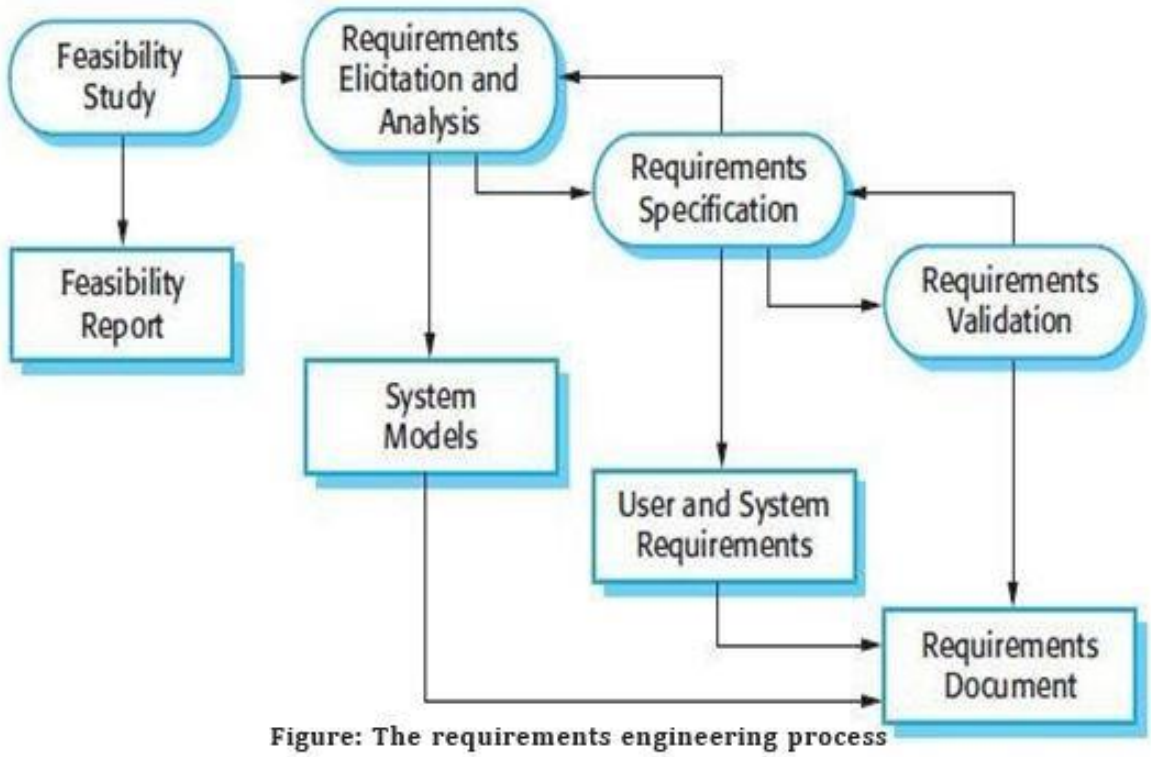


Figure: The requirements engineering process

Feasibility study:

An estimate is made of whether the identified user needs may be satisfied using current software and hardware technologies.

The study considers whether the proposed system will be cost-effective from a business point of view and if it can be developed within existing budgetary constraints.

A feasibility study should be relatively cheap and quick.

The result should inform the decision of whether or not to go ahead with a more detailed analysis.

- A feasibility study is a short, focused study that should take place early in the RE process.
- It should answer three key questions:
  a) does the system contribute to the overall objectives of the organization?
  b) can the system be implemented within schedule and budget using current technology? And
  c) can the system be integrated with other systems that are used?

  If the answer to any of these questions is no, you should probably not goahead with the project.

Requirements elicitation and analysis:

After an initial feasibility study, the next stage of the requirements engineering process is requirements elicitation and analysis.

In this activity, software engineers work with customers and system end-users to find out about the application domain, what services the system should provide, the required performance of the system, hardware constraints, and so on.

Requirements elicitation and analysis may involve a variety of different kinds of people in an organization.

A system stakeholder is anyone who should have some direct or indirect influence on the system requirements. Stakeholders include end users who will interact with the system and anyone else in an organization who will be affected byit. Other system stakeholders might be engineers who are developing or maintaining other related systems, business managers, domain experts, and trade union representatives.

A process model of the elicitation and analysis process is shown in Figure below.

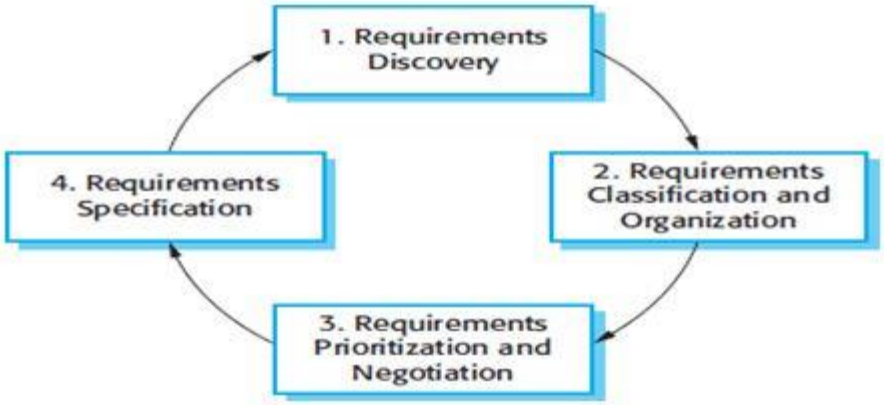


Figure: The requirements elicitation and analysis process

Each organization will have its own version or instantiation of this general model depending on local factors such as the expertise of the staff, the type of system being developed, the standards used, etc.

The process activities are:

1. Requirements discovery:

   This is the process of interacting with stakeholders of the system to discover their requirements. Domain requirements from stakeholders and documentation are also discovered during this activity.

2. Requirements classification and organization:

   This activity takes the unstructured collection of requirements, groups related requirements, and organizes them into coherent clusters. The most common way of grouping requirements is to use a model of the system architecture to identify sub-systems and to associate requirements with each sub-system. In practice, requirements engineering and architectural design cannot be completely separate activities.

3. Requirements prioritization and negotiation:

   Inevitably, when multiple stakeholders are involved, requirements will conflict. This activity is concerned with prioritizing requirements and finding andresolving requirements conflicts through negotiation. Usually, stakeholders have to meet to resolve differences and agree on compromise requirements.

4. Requirements specification:

   The requirements are documented and input into the next round of the spiral. Formal or informal requirements documents may be produced.

The above figure shows that requirements elicitation and analysis is an iterative process with continual feedback from each activity to other activities.

The process cycle starts with requirements discovery and ends with the requirements documentation. The analyst's understanding of the requirements improves with each round of the cycle. The cycle ends when the requirements document is complete.

Eliciting and understanding requirements from system stakeholders is a difficult process for several reasons:

1. Stakeholders often don't know what they want from a computer system except in the most general terms; they may find it difficult to articulate what they want the system to do; they may make unrealistic demands because they don't know what is and isn't feasible.
2. Stakeholders in a system naturally express requirements in their own terms and with implicit knowledge of their own work. Requirements engineers, without experience in the customer's domain, may not understand these requirements.
3. Different stakeholders have different requirements and they may express these in different ways. Requirements engineers have to discover all potential sources of requirements and discover commonalities and conflict.
4. Political factors may influence the requirements of a system. Managers may demand specific system requirements because these will allow them to increase their influence in the organization.
5. The economic and business environment in which the analysis takes place is dynamic. It inevitably changes during the analysis process. The importance of particular requirements may change. New requirements may emerge from new stakeholders who were not originally consulted.

Requirements specification:

Requirements specification is the activity of translating the information gathered during the analysis activity into a document that defines a set of requirements. Two types of requirements may be included in this document. User requirements are abstract statements of the system requirements for the customer and end-user of the system; system requirements are a more detailed description of the functionality to be provided.

Requirements validation:

Requirements validation is the process of checking that requirements actually define the system that the customer really wants.

It overlaps with analysis as it is concerned with finding problems with the requirements.

Requirements validation is important because errors in a requirements document can lead to extensive rework costs when these problems are discovered during development or after the system is in service.

The cost of fixing a requirements problem by making a system change is usually much greater than repairing design or coding errors.

During the requirements validation process, different types of checks should be carried out on the requirements in the requirements document. These checks include:

1. Validity checks:

   A user may think that a system is needed to perform certain functions. However, further thought and analysis may identify additional or different functions that are required. Systems have diverse stakeholders with different needs and any set of requirements is inevitably a compromise across the stakeholder community.

2. Consistency checks:

   Requirements in the document should not conflict. That is, there should not be contradictory constraints or different descriptions of the same system function.

3. Completeness checks:

   The requirements document should include requirements that define all functions and the constraints intended by the system user.

4. Realism checks:

   Using knowledge of existing technology, the requirements should be checked to ensure that they can actually be implemented. These checks should also take account of the budget and schedule for the system development.

5. Verifiability:

   To reduce the potential for dispute between customer and contractor, system requirements should always be written so that they are verifiable. This means that you should be able to write a set of tests that can demonstrate that the delivered system meets each specified requirement.

There are a number of requirements validation techniques that can be usedindividually or in conjunction with one another:

1. Requirements reviews:

The requirements are analyzed systematically by a team of reviewers who check for errors and inconsistencies.

2. Prototyping:

In this approach to validation, an executable model of the system in questionis demonstrated to end-users and customers. They can experiment with this modelto see if it meets their real needs.

3. Test-case generation:

Requirements should be testable. If the tests for the requirements are devised as part of the validation process, this often reveals requirements problems. If a testis difficult or impossible to design, this usually means that the requirements will be difficult to implement and should be reconsidered. Developing tests from the user requirements before any code is written is an integral part of extreme programming.

Requirements management:

Once a system has been installed and is regularly used, new requirements inevitably emerge due to business, organizational, and technical changes which lead to changes to the requirements for a software system.

Requirements management is the process of understanding and controlling changes to system requirements.

You need to establish a formal process for making change proposals andlinking these to system requirements.

The formal process of requirements management should start as soon as a draft version of the requirements document is available.

However, you should start planning how to manage changing requirements during the requirements elicitation process.

Requirements management planning:

Planning is an essential first stage in the requirements management process. Itestablishes the level of requirements management detail that is required.

During this requirements management stage, you have to decide on:

1. Requirements identification:

   Each requirement must be uniquely identified so that it can be cross- referenced with other requirements and used in traceability assessments.

2. A change management process:

   This is the set of activities that assess the impact and cost of changes.

3. Traceability policies:

   These policies define the relationships between each requirement and between the requirements and the system design that should be recorded. The traceability policy should also define how these records should be maintained.

4. Tool support:

   Requirements management involves the processing of large amounts of information about the requirements. Tools that may be used range from

specialist requirements management systems to spreadsheets and simple database systems.

Requirements management needs automated support and the software tools for thisshould be chosen during the planning phase. You need tool support for:

1. Requirements storage:

The requirements should be maintained in a secure, managed data store thatis accessible to everyone involved in the requirements engineering process.

2. Change management:

The process of change management is simplified if active tool support isavailable.

3. Traceability management:

Tool support for traceability allows related requirements to be discovered. Some tools are available which use natural language processing techniques to help discover possible relationships between requirements.

For small systems, it may not be necessary to use specialized requirements management tools. It may be supported using the facilities available in word processors, spreadsheets, and PC databases. However, for larger systems, more specialized tool support is required.

Requirements change management:

Requirements change management should be applied to all proposed changes to a system's requirements after the requirements document has been approved.

Change management is essential because you need to decide if the benefits of implementing new requirements are justified by the costs of implementation.

The advantage of using a formal process for change management is that all change proposals are treated consistently and changes to the requirements document are made in a controlled way.

Figure: Requirements change management

There are three principal stages to a change management process:

1. Problem analysis and change specification:

   - The process starts with an identified requirements problem or, sometimes, with a specific change proposal.
   - During this stage, the problem or the change proposal is analyzed to check that it is valid. This analysis is fed back to the change requestor who may respond with a more specific requirements change proposal, or decide to withdraw the request.

2. Change analysis and costing:

   - The effect of the proposed change is assessed using traceability information and general knowledge of the system requirements.
   - The cost of making the change is estimated both in terms of modifications to the requirements document and, if appropriate, to the system design andimplementation.
   - Once this analysis is completed, a decision is made whether or not to proceed with the requirements change.

3. Change implementation:

   - The requirements document and, where necessary, the system design and implementation, are modified.
   - We should organize the requirements document so that we can make changes to it without extensive rewriting or reorganization.
   - As with programs, changeability in documents is achieved by minimizing external references and making the document sections as modular as possible. Thus, individual sections can be changed and replaced without affecting other parts of the document.

If a new requirement has to be urgently implemented, there is always a temptation to change the system and then retrospectively modify the requirements document.

oOo

UNIT III

DESIGN ENGINEERING

Syllabus

Design Engineering: Design process and design quality, design concepts, the design model. **Creating an architectural design:** software architecture, data design, architectural styles and patterns, architectural design, conceptual model of UML, basic structural modeling, class diagrams, sequence diagrams, collaboration diagrams, use case diagrams, component diagrams.

***

Design Engineering:

Design:

Design is a meaningful engineering representation of something that is to be built.

It can be traced to a customer's requirements and at the same time assessed for quality against a set of predefined criteria for "good" design.

In the software engineering context, design focuses on four major areas of concern: ***data, architecture, interfaces, and components.***

Software engineers design computer based systems, but the skills required at each level of design work are different.

- At the data and architectural level, design focuses on patterns as they apply to the application to be built.
- At the interface level, human ergonomics (human factors) often dictate our design approach.
- At the component level, a "programming approach" leads us to effective data and procedural designs.

Why is it important?

Design allows you to model the system or product that is to be built. This model can be assessed for quality and improved before code is generated, tests are conducted, and end users become involved in large numbers.

The goal of design is to produce a model or representation that exhibits firmness, commodity, and delight. To accomplish this, you must practice diversification and then convergence.

Design Engineering:

Design engineering encompasses the set of principles, concepts, and practices that lead to the development of a high-quality system or product.

◎Design principles establish an overriding philosophy that guides you in the design work you must perform.

◎Design concepts must be understood before the mechanics of design practice are applied.

◎Design practice itself leads to the creation of various representations of the software that serve as a guide for the construction activity that follows.

Design Within the Context of Software Engineering:

"The most common miracle of software engineering is the transition from analysis to design and design to code."

Once software requirements have been analyzed and modeled, software design is the last software engineering action within the modeling activity and sets the stage for construction (code generation and testing).

Each of the elements of the requirements model provides information that is necessary to create the four design models required for a complete specification of design. The flow of information during software design is illustrated in figure below.

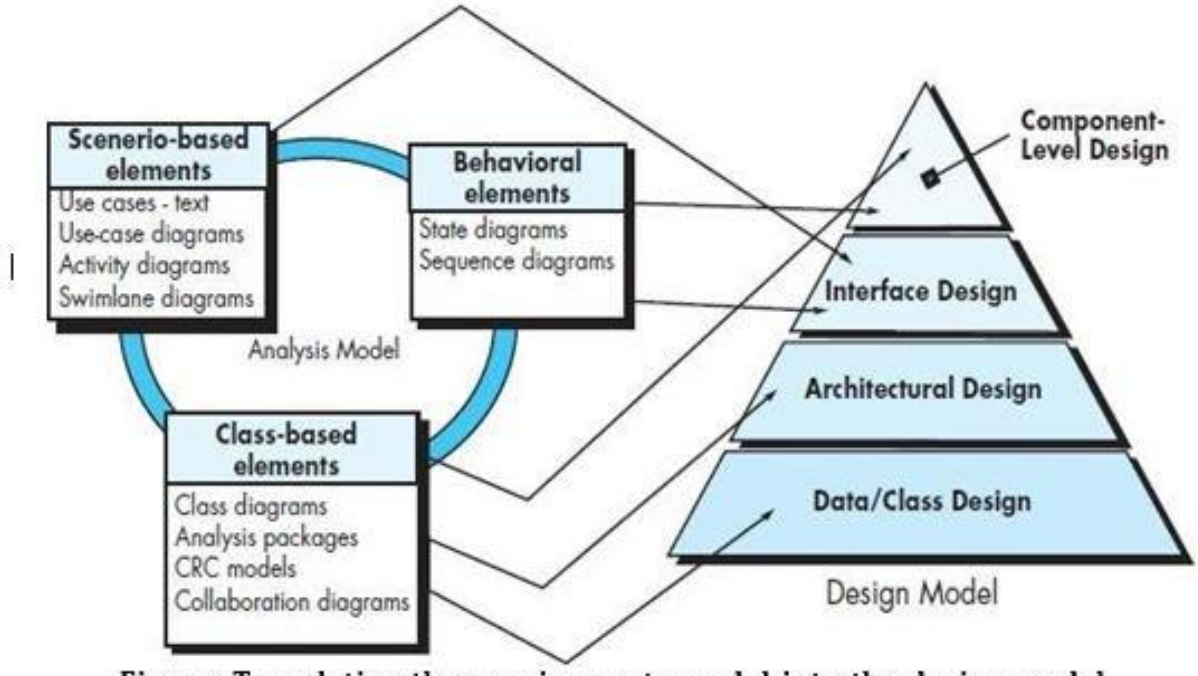


Figure: Translating the requirements model into the design model

The requirements model, manifested by scenario-based, class-based, flow- oriented, and behavioral elements, feed the design task. Using design notation and design methods discussed in later chapters, design produces a data/class design, an architectural design, an interface design, and a component design.

- The data/class design transforms class models into design class realizations and the requisite data structures required to implement the software.
- Part of class design may occur in conjunction with the design of software architecture.
- More detailed class design occurs as each software component is designed.
- The architectural design defines the relationship between major 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 architecture can be implemented.
- The interface design describes how the software communicates with systems that interoperate with it, and with humans who use it.
- The component-level design transforms structural elements of the software architecture into a procedural description of software components. Information obtained from the class-based models, flow models, and behavioral models serve as the basis for component design.

oOo

Design Process and Design Quality:

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

Throughout the design process, the quality of the evolving design is assessed with a series of formal technical reviews or design walkthroughs. McGlaughlin suggests three characteristics that 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 behavioural domains from an implementation perspective.

Each of these characteristics is actually a goal of the design process.

Quality Guidelines:

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

The following are the quality guidelines:

1. A design should exhibit an architecture that
   a) has been created using recognizable architectural styles or patterns,
   b) is composed of components that exhibit good design, and
   c) can be implemented in an evolutionary fashion, thereby facilitating implementation and testing.
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.

These design guidelines are not achieved by chance. They are achieved through the application of fundamental design principles, systematic methodology, and thorough review.

Quality Attributes:

Hewlett-Packard developed a set of software quality attributes that has been given the acronym FURPS—functionality, usability, reliability, performance, and supportability.

The FURPS quality attributes represent a target for all software design:

- **Functionality** is assessed by evaluating the feature 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 (MTTF), the ability to recover from failure, and the predictability of the program.
- **Performance** is measured by considering processing speed, response time, resource consumption, throughput, and efficiency.
- **Supportability** combines the ability to extend the program (extensibility), adaptability, serviceability - these three attributes represent a more common term, maintainability and in addition, testability, compatibility, configurability (the ability to organize and control elements of the software configuration), the ease with which a system can be installed, and the ease with which problems can be localized.

Not every software quality attribute is weighted equally as the software design is developed.

- One application may stress functionality with a special emphasis on security.
- Another may demand performance with particular emphasis on processing speed.
- A third might focus on reliability.

Regardless of the weighting, it is important to note that these quality attributes must be considered as design commences, *not* after the design is complete and construction has begun.

Design Concepts:

A set of fundamental software design concepts has evolved over the history of software engineering.

Although the degree of interest in each concept has varied over the years, each has stood the test of time. Each provides the software designer with a foundation from which more sophisticated design methods can be applied.

M. A. Jackson once said: "The beginning of wisdom for a [software engineer] is to recognize the difference between getting a program to work, and getting it right." Fundamental software design concepts provide the necessary framework for "getting it right."

Following are the important software design concepts that span both traditional and object-oriented software development.

Abstraction:

Each step in the software process is a refinement in the level of abstraction of the software solution.

Many levels of abstraction are there.

- 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 is a named sequence of instructions that has a specific and limited function.

An example of a procedural abstraction would be the word open for a door. Open implies a long sequence of procedural steps (e.g., walk to the door, reach out and grasp knob, turn knob and pull door, step away from moving door, etc.).

- **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, opening mechanism, weight, dimensions).

Architecture:

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

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

One goal of software design is to derive an architectural rendering of a system. This rendering serves as a framework from which more detailed design activities are conducted. A set of architectural patterns enables 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 (patterns) 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 the business or technical process that the system must accommodate.
- Finally, **functional models** can be used to represent the functional hierarchy of a system.

Patterns:

Brad Appleton defines a design pattern in the following manner: "a pattern is a named nugget of inside which conveys that essence of a proven solution to a recurring problem within a certain context amidst competing concerns."

A design pattern describes a design structure that solves a particular design 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

- Whether the pattern is capable to the current work,
- Whether the pattern can be reused,
- Whether the pattern can serve as a guide for developing a similar, but functionally or structurally different pattern.

Modularity:

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

It has been stated that "modularity is the single attribute of software that allows a program to be intellectually manageable". Monolithic software 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.

The "divide and conquer" strategy- it's easier to solve a complex problem when you break it into manageable pieces. This has important implications with regard to modularity and software. If we subdivide software indefinitely, the effort required to develop it will become negligibly small. The effort to develop an individual software module does decrease as the total number of modules increases. Given the same set of requirements, more modules means smaller individual size. However, as the number of modules grows, the effort associated with integrating the modules also grow.

Under modularity or over modularity should be avoided. We modularize a design so that development can be more easily planned; software increment can be defined and delivered; changes can be more easily accommodated; testing and debugging can be conducted more efficiently, and long-term maintenance can be conducted without serious side effects.

Information Hiding:

The principle of information hiding suggests that modules be "characterized by design decision that hides from all others."

Modules should be specified and designed so that information 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 a set of independent modules that communicate with one another only that information necessary to achieve software function. Abstraction helps to define the procedural entities that make up the software. Hiding defines and enforces access constraints to both procedural detail within a module and local data structure used by module.

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 procedure are hidden from other parts of the software, inadvertent errors introduced during modification are less likely to propagate to other locations within software.

Functional Independence:

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

Functional independence is achieved by developing modules with "single minded" function and an "aversion" to excessive interaction with other modules. Stated another way, we want to 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.

Software with effective modularity, that is, independent modules, is easier to develop because function may be compartmentalized and interfaces are simplified. Independent sign or code modifications are limited, error propagation is reduced, and reusable modules are possible. To summarize, 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. Cohesion is a natural extension of the information hiding.

A cohesive module performs a single task, requiring little interaction with other components in other parts of a program. Stated simply, a cohesive module should do just one thing.

Coupling is an indication of interconnection among modules in a software structure. Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface. In software design, we strive for lowest possible coupling. Simple connectivity among modules results in software that is easier to understand and less prone to a "ripple effect", caused when errors occur at one location and propagates throughout a system.

Refinement:

Stepwise refinement is a top- down design strategy originally proposed by Niklaus Wirth.

A program is developed by successively refining levels of procedural detail. A hierarchy is developed by decomposing a macroscopic statement of function in a step wise fashion until programming language statements are reached.

Refinement is actually a process of elaboration. We begin with a statement of function that is defined at a high level of abstraction. That is, 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 occurs.

Abstraction and refinement are complementary concepts. 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. Both concepts aid the designer in creating a complete design model as the design evolves.

Refactoring :

Refactoring is a reorganization technique that simplifies the design of a component without changing its function or behavior.

Fowler 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 refactored, 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. The designer may decide that the component should be refactored into 3 separate components, each exhibiting high cohesion. The result will be software that is easier to integrate, easier to test, and easier to maintain.

Design classes:

The software team must define a set of design classes that

- Refine the analysis classes by providing design detail that will enable the classes to be implemented, and
- Create a new set of design classes that implement a software infrastructure to support the design solution.

Five different types of design classes, each representing a different layer of the design architecture are suggested.

User interface classes: define all abstractions that are necessary for human computer interaction. In many cases, HCL occurs within the context of a metaphor and the design classes for the interface may be visual representations of the elements of the metaphor.

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.

Process classes implement lower – level business abstractions required to fully manage the business domain classes.

Persistent classes represent data stores that will persist beyond the execution of the software.

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.

As the design model evolves, the software team must develop a complete set of attributes and operations for each design class. The level of abstraction is reduced as each analysis class is transformed into a design representation.

Arlow and Neustadt suggest that each design class be reviewed to ensure that it is "well-formed." They define four characteristics of a well- formed design class:

- **Complete and sufficient:** A design class should be the complete encapsulation of all

attributes and methods that can reasonably be expected to exist for the class. Sufficiency ensures that the design class contains only those methods that are sufficient to achieve the intent of the class, no more and no less.

- **Primitiveness:** Methods associated with a design class should be focused on accomplishing one service for the class. Once the service has been implemented with a method, the class should not provide another way to accomplish the same thing.
- **High cohesion:** A cohesive design class has a small, focused set of responsibilities and single- mindedly applies attributes and methods to implement those responsibilities.
- **Low coupling:** Within the design model, it is necessary for design classes to collaborate with one another. However, collaboration should be kept to an acceptable minimum. If a design model is highly coupled the system is difficult to implement, to test, and to maintain over time. In general, design classes within a subsystem should have only limited knowledge of classes in other subsystems. This restriction, called the law of Demeter, suggests that a method should only sent messages to methods in neighboring classes.

The Design Model:

The design model can be viewed in two different dimensions.

- The process dimension indicates the evolution of the design model as design tasks are executed as a part of the software process.
- The abstraction dimension represents the level of detail as each element of the analysis model is transformed into a design equivalent and then refined iteratively.

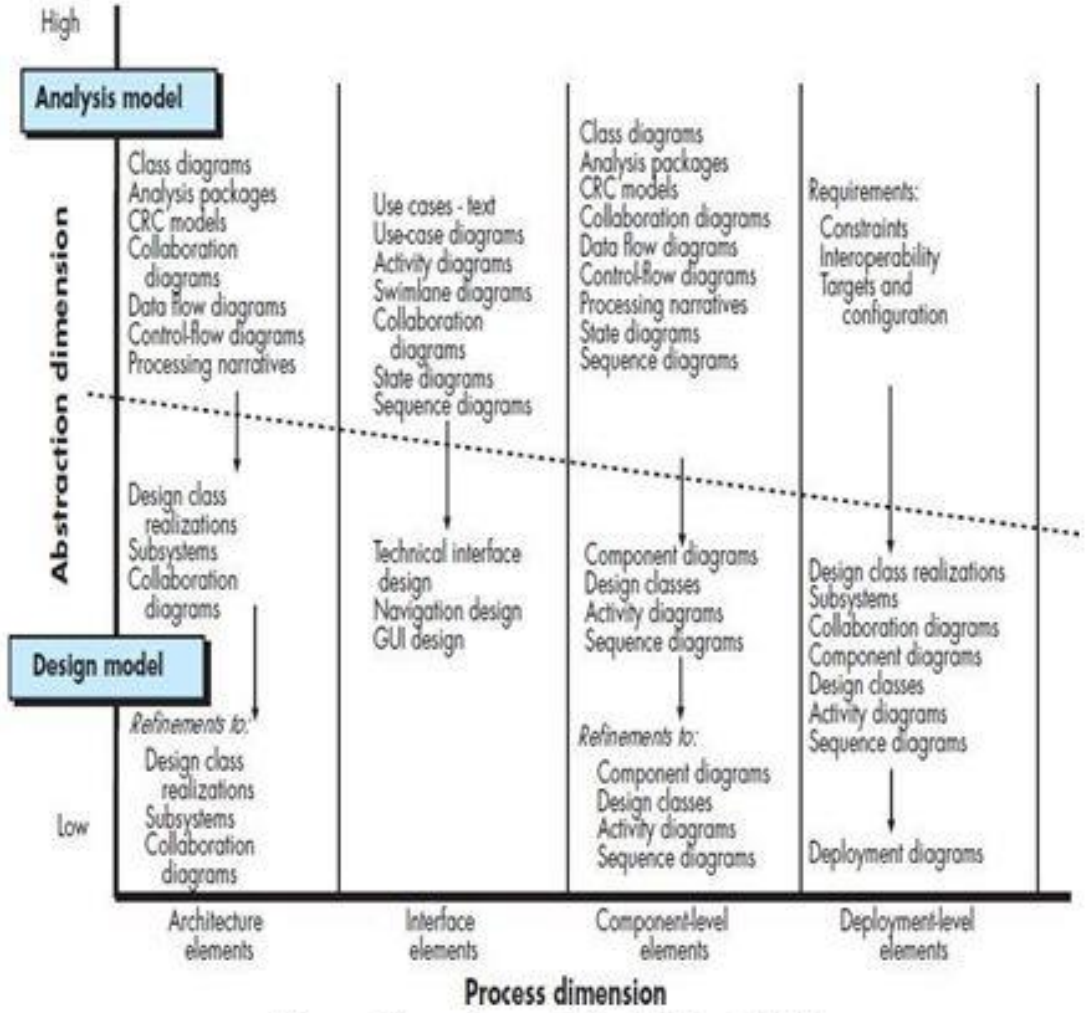


Figure: Dimensions of the Design Model

The elements of the design model use many of the same UML diagrams that were used in the analysis model. The difference is that these diagrams are refined and elaborated as a path of design; more implementation- specific detail is provided, and architectural structure and style, components that reside within the architecture, and the interface between the components and with the outside world are all emphasized.

It is important to mention however, that model elements noted along the horizontal axis are not always developed in a sequential fashion. In most cases preliminary architectural design sets the stage and is followed by interface design and component-level design, which often occur in parallel. The deployment model us usually delayed until the design has been fully developed.

Data design elements:

Data design sometimes referred to as data architecting creates a model of data and/or information that is represented at a high level of abstraction. This data model is then refined into progressively more implementation-specific representations that can be processed by the computer-based system.

The structure of data has always been an important part of software design.

- At the program component level, the design of data structures and the associated algorithms required to manipulate them is essential to the criterion of high-quality applications.
- At the application level, the translation of a data model into a database is pivotal to achieving the business objectives of a system.
- At the business level, the collection of information stored in disparate databases and reorganized into a "data warehouse" enables data mining or knowledge discovery that can have an impact on the success of the business itself.

Architectural design elements:

The architectural design for software is the equivalent to the floor plan of a house.

The architectural model is derived from three sources.

- Information about the application domain for the software to be built.
- Specific analysis model elements such as data flow diagrams or analysis classes, their relationships and collaborations for the problem at hand, and
- The availability of architectural patterns

Interface design elements:

The interface design for software is the equivalent to a set of detailed drawings for the doors, windows, and external utilities of a house.

The interface design elements for software tell how information flows into and out of the system and how it is communicated among the components defined as part of the architecture.

There are 3 important elements of interface design:

- The user interface (UI);
- External interfaces to other systems, devices, networks, or other produces or consumers of information; and
- Internal interfaces between various design components.

These interface design elements allow the software to communicated externally and enable internal communication and collaboration among the components that populate the software architecture.

UI design is a major software engineering action.

The design of a UI incorporates aesthetic elements (e.g., layout, color, graphics, interaction mechanisms), ergonomic elements (e.g., information layout and placement, metaphors, UI navigation), and technical elements (e.g., UI patterns, reusable components). In general, the UI is a unique subsystem within the overall

application architecture.

The design of external interfaces requires definitive information about the entity to which information is sent or received. The design of external interfaces should incorporate error checking and appropriated security features.

UML defines an interface in the following manner:" an interface is a specifier for the externally- visible operations of a class, component, or other classifier without specification of internal structure."

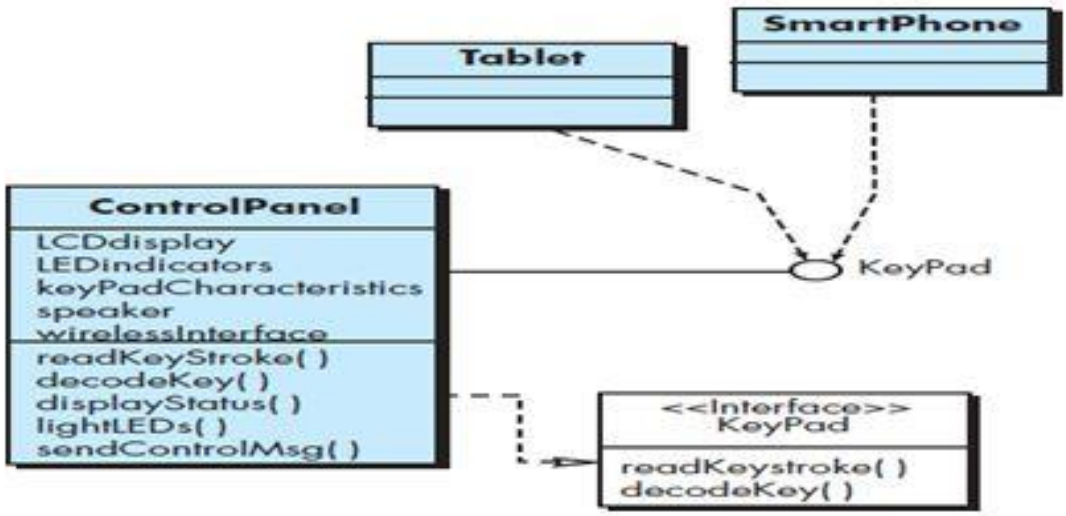


Figure: UML Interface representation of ControlPanel

COMPONENT-LEVEL DESIGN ELEMENTS

The component-level design for software is equivalent to a set of detailed drawings.

The component-level design for software fully describes the internal detail of each software component. To accomplish this, the component-level design defines data structures for all local data objects and algorithmic detail for all processing that occurs within a component and an interface that allows access to all component operations.

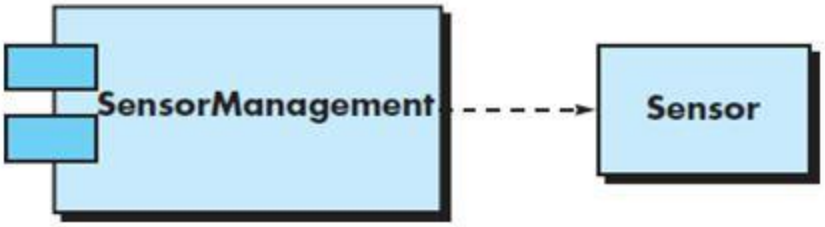


Figure: UML Component Diagram

Deployment-level design elements:

Deployment-level design elements indicated how software functionality and subsystems will be allocated within the physical computing environment that will support the software

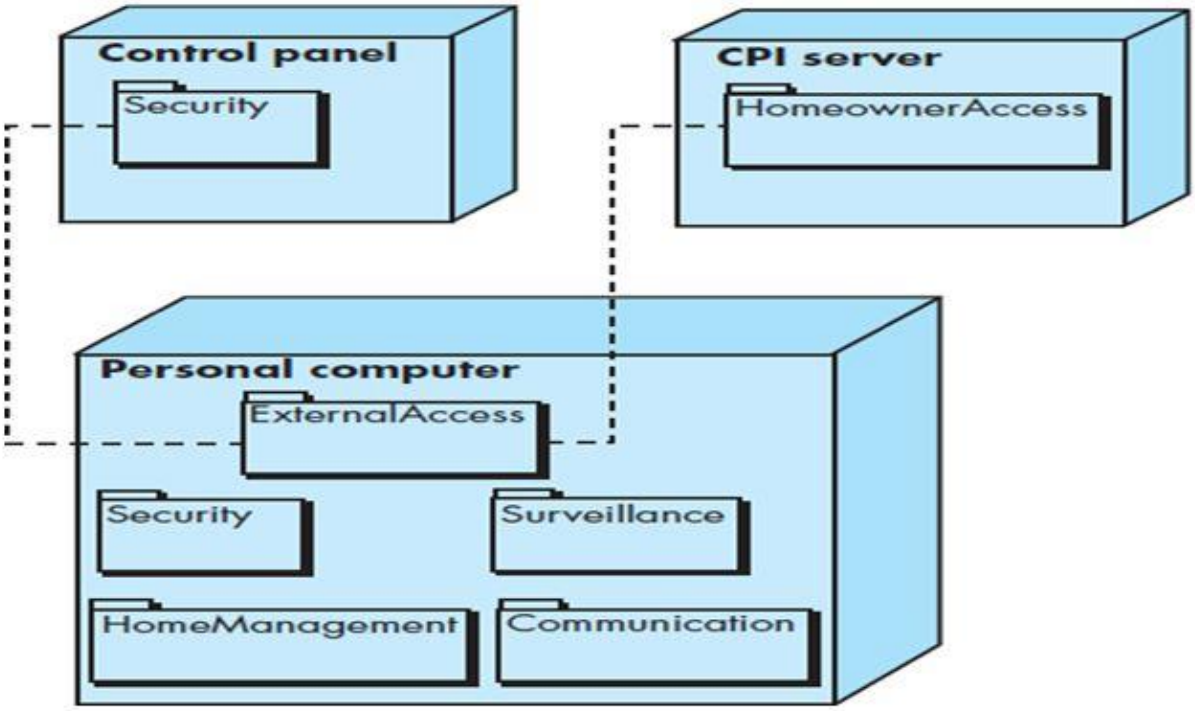


Figure: UML Deployment Diagram

Creating an Architectural Design

The architectural design is the preliminary blueprint from which software is constructed.

Software Architecture :

Today, effective software architecture and its explicit representation and design have become dominant themes in software engineering.

➢ What is Architecture?

"The architecture of a system is a comprehensive framework that describes its form and structure - its components and how they fit together."

In simple words, architecture captures system structure in terms of components and how they interact.

Architecture is defined as the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution.

➢ What is Software Architecture?

The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.

The architecture is not the operational software. Rather, it is a representation that enables you to

1. Analyze the effectiveness of the design in meeting its stated requirements,
2. Consider architectural alternatives at a stage when making design changes is still relatively easy, and
3. Reduce the risks associated with the construction of the software.

➢ Why Is Architecture Important?

There are three key reasons stating that software architecture is important:

1. Representations of software architecture are an enabler for communication between all parties (stakeholders) interested in the development of a computer-based system.
2. The architecture highlights early design decisions that will have a profound impact on all software engineering work that follows and, as important, on the ultimate success of the system as an operational entity.

3. Architecture "constitutes a relatively small, intellectually graspable model of how the system is structured and how its components work together".

Design of Software Architecture :

The design of software architecture considers two levels of the design pyramid:

1. Data Design
2. Architectural Design

Data design enables us to represent the data components of the architecture in conventional systems and class definitions (encapsulating attributes and operations) in object-oriented systems.

Architectural design focuses on the representation of the structure of software components, their properties, and the interactions.

➢ Data Design :

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

➢ Data Design at the Architectural Level :

Today, businesses have dozens of databases serving many applications encompassing hundreds of gigabytes of data. The Challenge is to extract useful information from this data environment, particularly when the information desiredis cross-functional (e.g., information that can be obtained only if specific marketing data are cross-correlated with product engineering data).

To solve this challenge, the business IT community has developed data mining techniques, also called *knowledge discovery in databases (KDD)*, that navigate through existing databases in an attempt to extract appropriate business- level information. However, the existence of multiple databases, their different structures, the degree of detail contained with the databases, and many other factors make data mining difficult within an existing database environment.

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 applications but encompasses all data used by a business. In a sense, a data warehouse is a large, independent database that has access to the data that are stored in databases that serve the set of

applications required by a business.

➢ Data Design at the Component Level :

Data design at the component level focuses on the representation of data structures that are directly accessed by one or more software components. Wasserman has proposed a set of principles that may be used to specify and design such data structures. Following are the set of principles for data specification :

1. **The systematic analysis principles applied to function and behavior should also be applied to data.** Representations of dat flow and content should also be developed and reviewed. Data objects should be identified, alternative data organizations should be considered, and the impact of data modeling on software design should be evaluated.
2. **All data structues and the operations to be performed on each should be identified.** The design of an efficient data structure must take the operations to be performed on the data structure into account. The attributes and operations encapsulated within a class satisfy this principle.
3. **A mechanism for defining the content of each data object should be established and used to define both data and the operations applied to it.** Class diagrams define the data items(attributes) contained within a class and the processing(operations) that are applied to these data items.
4. **Low-level data design decisions should be deferred until late in the design process.** A process of stepwise refinement may be used for the design of data. That is, overall data organization may be defined during requirements analysis, refined during data design work, and specified in detail during component-level design.

   The represenattion of a data structure shold be known only to those modules that must make direct use of the data contained within the **structure.** The concept of information hiding and the related concept of coupling provide important insight into the quality of a software design.
5. **A library of useful data structures and the operations that may be applied to them should be developed.** A class library achieves this.
6. **A software design and programming language should support the specification and realization of abstract data types.** The implementation of sophisticated data structure can be made exceedingly difficult if no means for direct specification of the structure exists in the programming language

chosen for implementation.

These priciples form a basis for a component-level data design approach that canbe integrated into both the analysis and design activities.

Architecture Styles and Patterns:

The architectural style is also a template for construction. The software thatis built for computer-based systems also exhibits one of many architectural styles. Each style describes a system category that encompasses

1. A set of computers that perform a function required by a system.
2. A set of connectors that enable "communication, coordination, andcooperation" among components
3. Constraints that define how components can be integrated to form the system
4. Semantic models that enable a designer to understand the overall propertiesof a system by analyzing the known properties of its constituent parts.

An architectural style is a transformation that is imposed on the design of an entire system. The intent is to establish a structure for all components of the system. In the case where an existing architecture is to be reengineered the imposition of an architectural style will result in fundamental changes to the structure of the software including a reassignment of the functionality of components.

An architectural pattern, like an architectural style, imposes a transformation on the design of an architecture. However, a pattern differs from a style in a number of fundamental ways:

(1) the scope of a pattern is less broad, focusing on one aspect of the architecture rather than the architecture in its entirety;

(2) a pattern imposes a rule on the architecture, describing how the software will handle some aspect of its functionality at the infrastructure level (e.g., concurrency)

(3) architectural patterns tend to address specific behavioral issues within the context of the architecture (e.g., how real-time applications handle synchronization or interrupts).

Patterns can be used in conjunction with an architectural style to shape the overall structure of a system.

A Brief Taxonomy of Architectural Styles:

- **Data-centered architecture:**

A data store (e.g., a file or database) resides at the center of this architecture and is accessed frequently by other components that update, add, delete, orotherwise modify data within the store. Following figure illustrates a typical data-centered style.

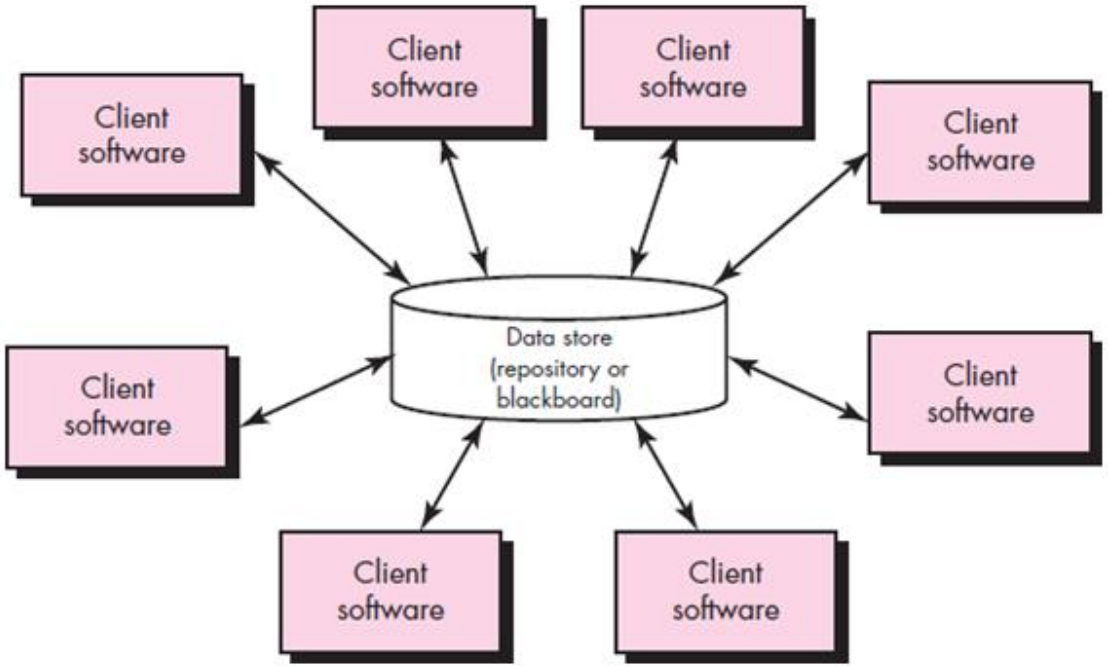


Figure: Data-centered Architecture

Client software accesses a central repository. In some cases the data repository is passive. That is, client software accesses the data independent of any changes to the data or the actions of other client software. A variation onthis approach transforms the repository into a "blackboard" that sends notifications to client software when data of interest to the client changes.

Data-centered architectures promote *integrability*.

- Data-flow architecture:

This architecture is applied when input data are to be transformed through a series of computational or manipulative components into output data. A pipe-and-filter pattern has a set of components, called *filters*, connected by *pipes* that transmit data from one component to the next. Each filter works independentlyof those components upstream and downstream, is designed to expect data input of a certain form, and produces data output (to the next

filter) of a specified form. However, the filter does not require knowledge of the workings ofits neighboring filters. If the data flow degenerates into a single line of transforms, it is termed batch sequential. This structure accepts a batch of data and then applies a series of sequential components (filters) to transform it.

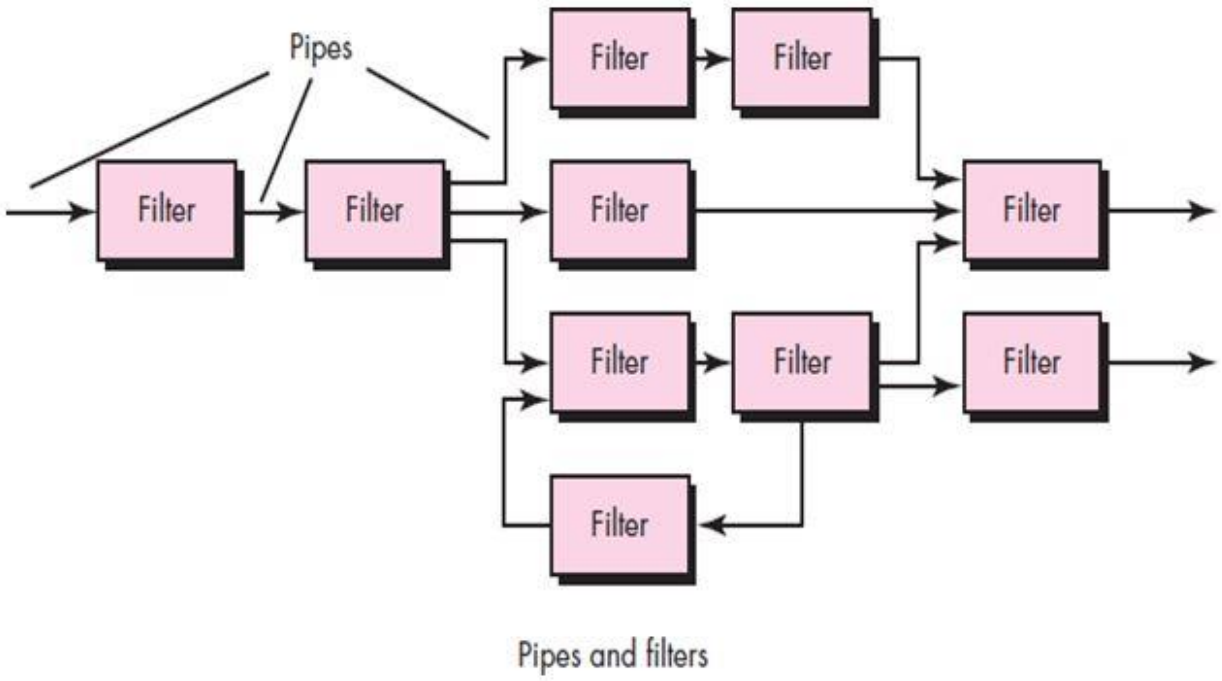


Figure: Data-flow Architecture

- **Call and return architecture:**

This architectural style enables you to achieve a program structure that is relatively easy to modify and scale. A number of substyles exist within this category:

1. ***Main program/subprogram architecture:*** This classic program structure decomposes function into a control hierarchy where a "main" program invokes a number of program components that in turn may invoke still other components. Following figure illustrates architecture of this type.

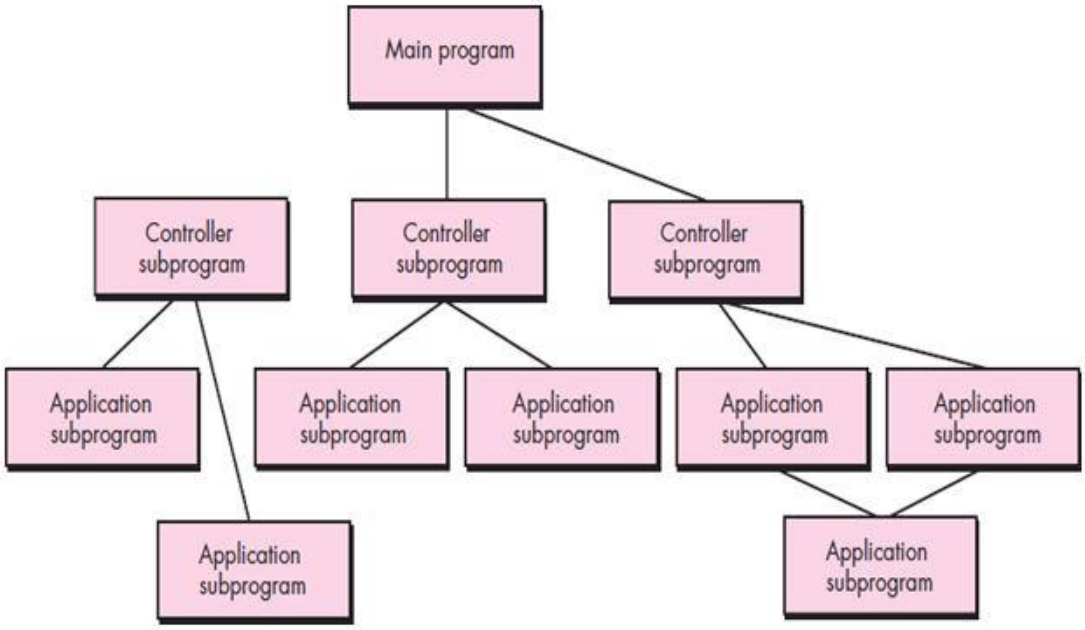


Figure: Main program/Sub program Architecture

2. ***Remote procedure call architecture:*** The components of a main program/subprogram architecture are distributed across multiple computers on a network.

➢ Object-oriented architecture:

The components of a system encapsulate data and the operations that must be applied to manipulate the data. Communication and coordination between components are accomplished via message passing.

➢ Layered architecture:

The basic structure of a layered architecture is illustrated in the following figure

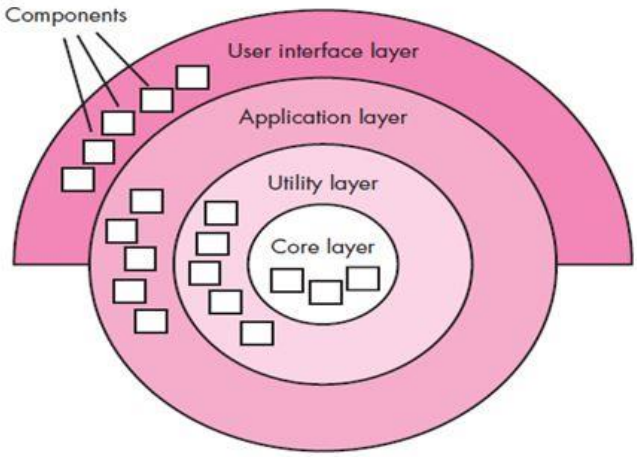


Figure: Layered Arhitecture

A number of different layers are defined, each accomplishing operations that progressively become closer to the machine instruction set. At the outer layer, components service user interface operations. At the inner layer, components perform operating system interfacing. Intermediate layers provide utility services and application software functions.

Architectural Patterns:

A software architecture may have a number of architectural patterns thataddress issues such as concurrency, persistence and distribution.

- **Concurrency:** Many applications must handle multiple tasks in a manner thatsimulates parallelism.

  For example:

  - ✓ *operating system process management* pattern
  - ✓ *task scheduler* pattern
- **Persistence:** Data persists if it survives past the execution of the process thatcreated it.

  For example:
  - ✓ a ***database management system*** pattern that applies the storage andretrieval capability of a DBMS to the application architecture
  - ✓ an ***application level persistence*** pattern that builds persistence features into the application architecture
- **Distribution:** The manner in which systems or components within systemscommunicate with one another in a distributed environment

For example: A ***broker*** acts as a 'middle-man' between the client componentand a server component. CORBA is an example of a broker architecture.

Organization and Refinement:

Because the design process often leaves you with a number of architectural alternatives, it is important to establish a set of design criteria that can be used to assess an architectural design that is derived. The following questions provide

insight into an architectural style:

Control:

- ✓ How is control managed within the architecture?
- ✓ Does a distinct control hierarchy exist, and if so, what is the role ofcomponents within this control hierarchy?
- ✓ How do components transfer control within the system?
- ✓ How is control shared among components?
- ✓ What is the control topology (i.e., the geometric form that the control takes)?
- ✓ Is control synchronized or do components operate asynchronously?

Data:

- ✓ How are data communicated between components?
- ✓ Is the flow of data continuous, or are data objects passed to the systemsporadically?
- ✓ What is the mode of data transfer (i.e., are data passed from one componentto another or are data available globally to be shared among system components)?
- ✓ Do data components (e.g., a blackboard or repository) exist, and if so, what is their role?
- ✓ How do functional components interact with data components?
- ✓ Are data components passive or active (i.e., does the data component actively interact with other components in the system)?
- ✓ How do data and control interact within the system?

These questions provide the designer with an early assessment of design quality and lay the foundation for more detailed analysis of the architecture.

Architectural Design:

As architectural design begins, the software to be developed must be put into context – that is, the design should define the external entities(other systesms, devices, people) that the software interacts with and the nature of the interaction. This information can generally be acquired from the analysis model and all other information gathered during requirements engineering.

Once context is modeled and all external software interfaces have been described, the designer specifies the structure of the system by defining and refining software components that implement the architecture.

Thies process continues iteratively until a complete architectural structure has been derived.

Representing the system in context:

At the architectural design level, a software architect uses an architectural context diagram(ACO) to model the manner in which software interacts with entities external to its boundaries.

The generic structure of the architectural context diagram is illustrated in figure.

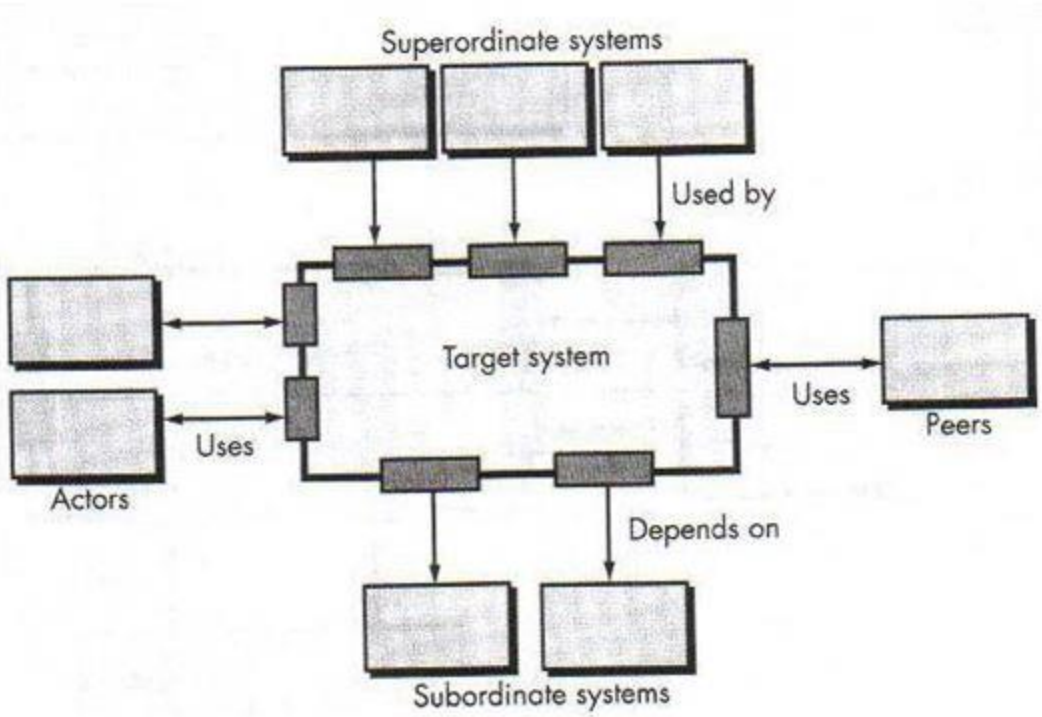


Figure: Architectural Context Diagram

Systems that interacts with the target system are represented as:

Superordinate systems: These are the systems which consider the (use the) target system inorder to complete its higher valued activities.

Subordinate systems: These are the systems which funtion along with the target system. Hence, supporting the target system in successfully completing its processing.

Peer systems or Peers: These are the systems which directly interact with the target system same as client-server interaction.

Actors: These are specimens or any entities possesing a definite set of roles and interacting with the system. During this interaction an actor can either provide or accept information from the system.

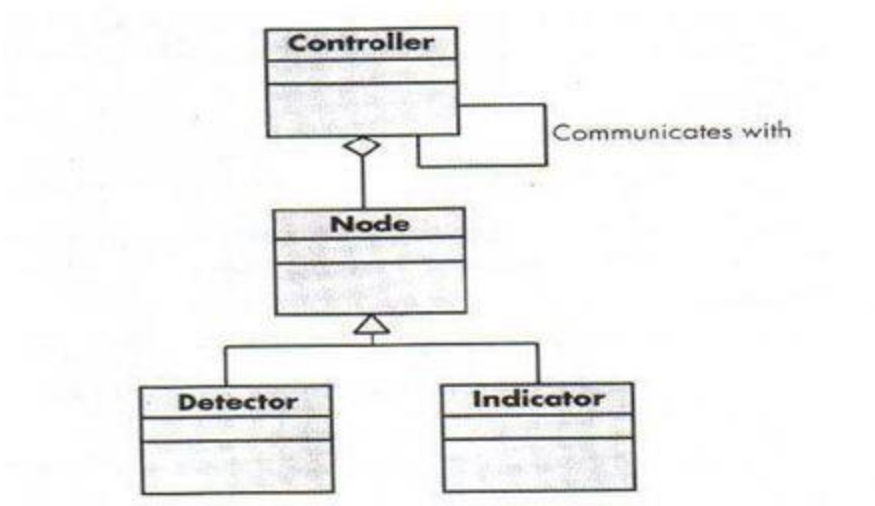
Each of these external entities communicates with the target system through an interface(the small shaded rectangles)

Following is the ACD(Architectural Context Diagram) depicting the safe home security systems.

Indicator – An abstraction that represents all mechanims for indicating that an alarm condition is occuring.(e.g. alarm siren, flash lights, bell).

Controller- An abstraction that depicts the mechanism that allows the arming or disarming of a node. If controller reside on a network, they have the ability to communicate with one another.

Each of these archetypes is depicted using UML notation as shown in figure:



UML Relationships for Safe Home Security function archetypes

As archetypes represents only abstractions. Hence, they can be further refined into components just by refining these abstractions. For example, detector might be refined into a class hierarchy of sensors.

Refining the Architecture into components:

As the software architecture is refined into components the structure of the system begins to emerge, for this purpose we initially consider the classes which were described as part of the analysis mode. These analysis classes forms the major entities of application domain. Hence, the application domain is one source for the derivation and refinement of components. Another source is the infrastructure domain. The architecture must accommodate components but

have no business connection to the application domain.

The interfaces depicted in the architecture context diagram imply one or more specialized components that process the data that fow across the interface. In some case graphical user interface, a complete subsystem architecture with many components must be designed.

- Components of the software architecture are dervied from three sources:
  - The application domain
  - The infrastructure domain
  - The interface domain

For safe home security system, we might define the set of top-level components as follows:

External Communication Management- Coordinates communication of the security function with external entities, for example, internet-based systesm, external alarm notification.

Control Panel Processing-manages all control panel functionality.

Detector Management- Coordinates access to all detectors attached to the system

Alarm Processing- verifies and acts on all alarm conditions.

Each of these top-level components would have to be elaborated iteratively and then positioned within the overall SafeHome architecture. Design classes(with appropriate attributes and operations) would be defined for each. It is important to note, how-ever, that the design details of all attributes and operations would not be specified until component-level design.

The overall architectural structure is illustrated in the following figure.

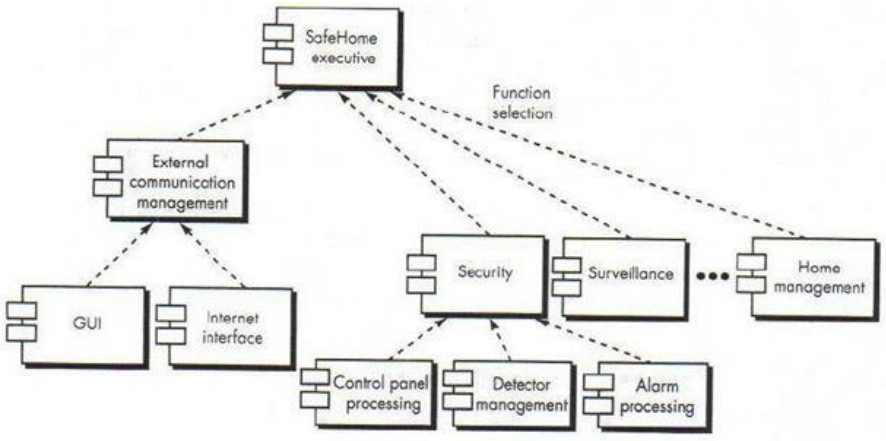


Figure: Overall architectural structure for Safe Home with top-level components

Transactions are acquired by external communication management as they move in from components that process the SafeHome GUI and the internet interface. This information is managed by a SafeHome executive componet that selects the appropriate product function. The control panel processing component interacts with homeowner to arm/disarm the security fucntion. The dector management component polls sensors to detect an alaram condition, and ahte alarm processing component produces output when alarm is detected.

Describing Instantiations of the System:

The architectural design that has been modeled to this point is still relatively high level. The context of the system has been represented, archetypes that indicate the important abstractions within the problem domain have been defined, the overall structure of the system is apparent, and the major software components have been identified. However, further refinement (recall that all design is iterative) is still necessary.

To accomplish this, an actual instantiation of the architecture is developed. By this I mean that the architecture is applied to a specific problem with the intent of demonstrating that the structure and components are appropriate.

The following figure illustrates an instantiation of the *SafeHome* architecture for the security system. Components shown in overall architecture are elaborated toshow additional detail. For example, the *detector management* component interacts with a *scheduler* infra-structure component that implements polling of each *sensor* object used by the security system.

Assessing Alternative Architectural Designs:

At its best, design results in a number of architectural alternatives that are each assessed to determine which is the most appropriate for the problem to be solved. In the sections that follow, we consider the assessment of alternatives architectural designs.

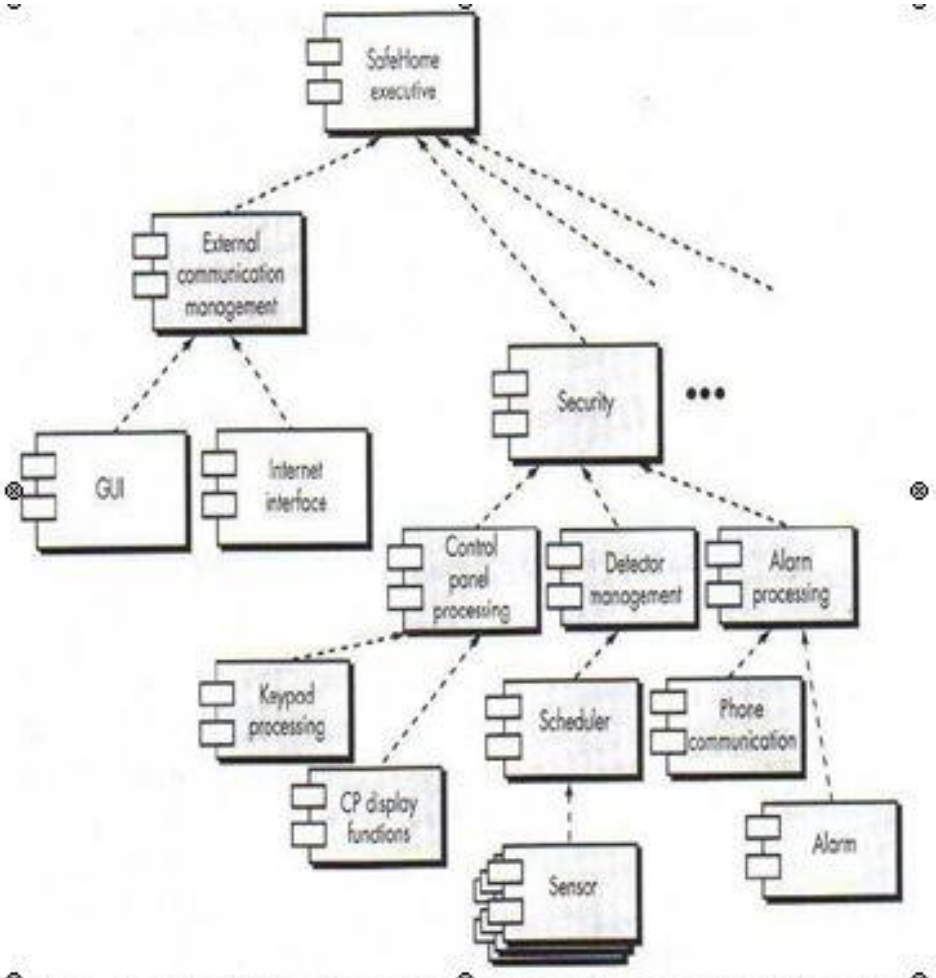


Figure: An Instantiation of security function with component elaboration

➢ An Architecture Trade-Off Analysis Method:

The Software Engineering Institute (SEI) has developed an *architecture trade-off analysis method* (ATAM) that establishes an iterative evaluation process for software architectures. The design analysis activities that follow are performed iteratively:

1. ***Collect scenarios.*** A set of use cases is developed to represent the system from the user's point of view.
2. ***Elicit requirements, constraints, and environment description.*** This information is determined as part of requirements engineering and is used to be certain that all stakeholder concerns have been addressed.
3. ***Describe the architectural styles/patterns that have been chosen to address the scenarios and requirements.*** The architectural style(s) should be described using one of the following architectural views:
   - ***Module view*** for analysis of work assignments with components

andthe degree to which information hiding has been achieved.
- ***Process view*** for analysis of system performance.
- ***Data flow view*** for analysis of the degree to which the architecture meets functional requirements.

4. ***Evaluate quality attributes by considering each attribute in isolation.*** The number of quality attributes chosen for analysis is a function of the timeavailable for review and the degree to which quality attributes are relevant to the sys-tem at hand. Quality attributes for architectural design assessment include reliability, performance, security, maintainability, flexibility, testability, portability, reusability, and interoperability.
5. ***Identify the sensitivity of quality attributes to various architectural attributes for a specific architectural style.*** This can be accomplished bymaking small changes in the architecture and determining how sensitive a quality attribute, say performance, is to the change. Any attributes that are significantly affected by variation in the architecture are termed *sensitivity points.*
6. ***Critique candidate architectures (developed in step 3) using the sensitivity analysis conducted in step 5.*** The SEI describes this approachin the following manner.

   Once the architectural sensitivity points have been determined, findingtrade-off points is simply the identification of architectural elements to

   which multiple attributes are sensitive. For example, the performance of a client-server architecture might be highly sensitive to the number of servers (performance increases, within some range, by increasing the number of servers). The number of servers, then, is a trade-off point with respect to this architecture.

These six steps represent the first ATAM iteration. Based on the results of steps 5 and 6, some architecture alternatives may be eliminated, one or more of the remaining architectures may be modified and represented in more detail and then the ATAM Steps are reapplied.

- Architectural Complexity:

A useful technique for assessing the overall complexity of a proposed architecture is to consider dependencies between components within the architecture. These dependencies are driven by information/control flow within the

system. Zhao suggests three types of dependencies:

Sharing dependencies represent dependence relationships among consumers who use the same resource or producers who produce for the same consumers. For example, for two components u and v, if u and v refer to the same global data, then there exists a shared dependence relationship between u and v.

Flow Dependencies represent dependence relationships between producers and consumers of resource. For example for two components u and v, if u must complete before controls flows into v or if u communicates with v by parameters, then there exists a flow dependence relationship between u and v.

Constrained dependencies represent constraints on the relative flow of control among a set of activities. For example, for two components u and v, if u andv cannot execute at the same time then there exists a constrained dependence relationship between u and v.

The sharing and flow dependencies noted by Zhao are similar to the concept of coupling. Coupling is an important design concept that is applicable at the architectural level and at the component level.

- Architectural Description Languages:

Architectural description language **(ADL)** provides a semantics and syntax for describing software architecture. ADL should provide the designer with the ability to decompose architectural components, compose individual components into larger architectural blocks, and represent interfaces (connection mechanisms) between components.

Once descriptive, language-based techniques for architectural design have been established, it is more likely that effective assessment methods for architectures will be established as the design evolves.