SOFTWARE ENGINEERING

KASNEB CICT PAPER NO.14

STUDY NOTES

SOFTWARE ENGINEERING KASNEB CICT PAPER 14

14.0 LEARNING OUTCOMES

A candidate who passes this paper should be able to:
• Identify appropriate software system design tools
• Design appropriate software systems
• Describe software system testing
• Document and commission software
• Evaluate software acquisition techniques
• Maintain software

CONTENT

14.1 Introduction to software systems development
- Software systems development concepts
- Software development life cycle

14.2 Software process models
- Linear/waterfall model
- Rapid prototyping
- Evolutionary models
- Component based models
- Other models

14.3 Software requirements analysis
- Overview or requirements concepts
- Requirement analysis process
- Requirements specification

14.4 Design tools and methods
- System flowcharts
- Case tools
- Functional decomposition
- Modules design
- Structured walkthrough
- Decision tables
- Structured charts
- Data flow diagrams
- Object Oriented design tools

14.5 Software quality
- Quality control and assurance
- Software quality factors and metrics
- Formal technical reviews
- Verification and validation
- Cost of quality

14.6 Software coding
- Coding styles and characteristics
- Coding in high-level languages
- Coding standards
- User interface

14.7 Software testing
- Software testing life cycle
- Software testing methods (Black box testing and White box testing)
- Software testing levels(unit integration, system and acceptance testing)
- Other forms of testing

14.8 Software acquisition methods
- Software costing
- Software outsourcing
- Open-source software engineering and customization
- In-house development
- Commercial Off The Shelf software (COTS)
- Budgeting for information systems
  - Financial cost benefit analysis
  - Business case approach
  - Total cost of ownership
  - Balanced scorecard/activity based costing and expected value
  - Tracking and allocating costs

14.9 Conversion strategies
- Conversion planning
- Parallel running
- Direct cut over
- Pilot study
- Phased approach

14.10 Documentation and commissioning
- Objectives of systems documentation
- Use of systems documentation
- Qualities of a good documentation
- Types of documentation
- Software commissioning

14.11 Software maintenance and evolution
- Types or software changes
- Software change identification
- Software change implementation

14.12 Auditing information systems
- Overview of information systems audit
- Auditing computer resources

- Audit techniques
- Audit applications

14.13 Emerging Issues and trends

TABLE OF CONTENTS

TOPIC 1
INTRODUCTION TO SOFTWARE ENGINEERING

- ***Software systems development concepts***

IEEE defines software engineering as:

The application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software; that is, the application of engineering to software.

Fritz Bauer, a German computer scientist, defines software engineering as:

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

Software Evolution

The process of developing a software product using software engineering principles and methods is referred to as **software evolution.** This includes the initial development of software and its maintenance and updates, till desired software product is developed, which satisfies the expected requirements.

Evolution starts from the requirement gathering process. After which developers create a prototype of the intended software and show it to the users to get their feedback at the early stage of software product development. The users suggest changes, on which several consecutive updates and maintenance keep on changing too. This process changes to the original software, till the desired software is accomplished.

Even after the user has desired software in hand, the advancing technology and the changing requirements force the software product to change accordingly. Re-creating software from scratch and to go one-on-one with requirement is not feasible. The only feasible and economical solution is to update the existing software so that it matches the latest requirements.

Software Evolution Laws

Lehman has given laws for software evolution. He divided the software into three different categories:

- **S-type (static-type) -** This is a software, which works strictly according to defined specifications and solutions. The solution and the method to achieve it, both are immediately understood before coding. The s-type software is least subjected to changes hence this is the simplest of all. For example, calculator program for mathematical computation.
- **P-type (practical-type) -** This is a software with a collection of procedures. This is defined by exactly what procedures can do. In this software, the specifications can be described but the solution is not obvious instantly. For example, gaming software.
- **E-type (embedded-type) -** This software works closely as the requirement of real-world environment. This software has a high degree of evolution as there are various changes in laws, taxes etc. in the real world situations. For example, Online trading software.

E-Type software evolution
Lehman has given eight laws for E-Type software evolution -
- **Continuing change -** An E-type software system must continue to adapt to the real world changes; else it becomes progressively less useful.
- **Increasing complexity -** As an E-type software system evolves, its complexity tends to increase unless work is done to maintain or reduce it.
- **Conservation of familiarity -** The familiarity with the software or the knowledge about how it was developed, why was it developed in that particular manner etc. must be retained at any cost, to implement the changes in the system.
- **Continuing growth-** In order for an E-type system intended to resolve some business problem, its size of implementing the changes grows according to the lifestyle changes of the business.
- **Reducing quality -** An E-type software system declines in quality unless rigorously maintained and adapted to a changing operational environment.
- **Feedback systems-** The E-type software systems constitute multi-loop, multi-level feedback systems and must be treated as such to be successfully modified or improved.
- **Self-regulation -** E-type system evolution processes are self-regulating with the distribution of product and process measures close to normal.
- **Organizational stability -** The average effective global activity rate in an evolving E-type system is invariant over the lifetime of the product.

Software Paradigms
Software paradigms refer to the methods and steps, which are taken while designing the software. There are many methods proposed and are in work today, but we need to see where in the software engineering these paradigms stand. These can be combined into various categories, though each of them is contained in one another:
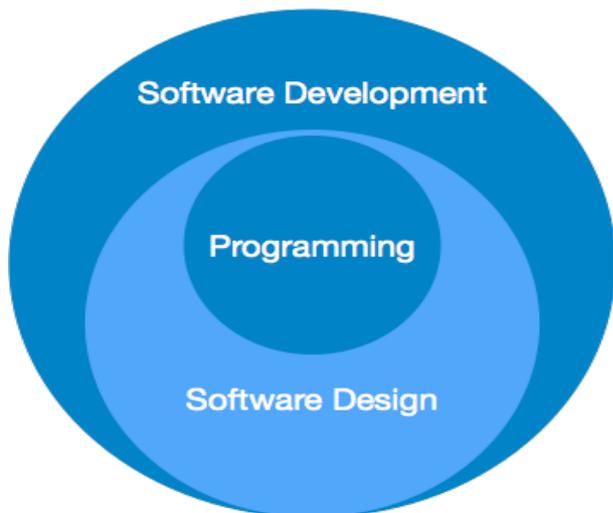
Programming paradigm is a subset of Software design paradigm which is further a subset of Software development paradigm.

Software Development Paradigm

This Paradigm is known as software engineering paradigms where all the engineering concepts pertaining to the development of software are applied. It includes various researches and requirement gathering which helps the software product to build. It consists of –

- Requirement gathering
- Software design
- Programming

Software Design Paradigm

This paradigm is a part of Software Development and includes :

- Design
- Maintenance
- Programming

Programming Paradigm

This paradigm is related closely to programming aspect of software development. This includes:

- Coding
- Testing
- Integration

Need of Software Engineering

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

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

always changing, new enhancements need to be done in the existing one. This is where software engineering plays a good role.

- **Quality Management-** Better process of software development provides better and quality software product.

Characteristics of good software

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

- Operational
- Transitional
- Maintenance

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

Operational

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

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

Transitional

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

- Portability
- Interoperability
- Reusability
- Adaptability

Maintenance

This aspect briefs about how well software has the capabilities to maintain itself in the ever-changing environment:

- Modularity
- Maintainability
- Flexibility
- Scalability

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

- ### *Software development life cycle*

SDLC is an approach for making software for the developer, user and customer. SDLC focus on the internal phase to the end phase for making particular software. It generally deals with the analyst and the corresponding clients. SDLC has some specific phase. This are-

1) Project identification

2) Feasibility study

3) System analysis

4) System design

5) System development

6) System testing

7) System implementation

8) System maintenance

9) System documentation

1) Project identification: - in this phase the analyst focus the basic objective and identification need for the corresponding software. In this phase the analyst set up some meeting with the corresponding client for making the desired software.
2) Feasibility study: - feasibility defines in the three views for making particular software for the client.
   - Technical
   - financial
   - Social feasibility.
3) System analysis: - analysis defines how and what type of desired software we have to make for the client. It has some pen and paper base. Exercise through which the analyst focused for the desired goals.
4) System design: - in this phase the analyst draw the corresponding diagrams related to the particular software. In this phase the design include in the form of flow chat, data flow diagram, net relationship diagram (NRD).
5) System development: - development refers in the form of coding, error checking and debarking for the particular software. This phase deals with the developer activity for making a successfully software.
6) System testing: - testing refers whatever analyst and developer done will it be correct and error free to the desired software. In the S.E there some testing technique to which we can check whether project is error free.
 The main testing techniques are
   o white box testing

   o black box testing

   o ad hope testing

   o system testing

   o unit testing

   o alpha testing

   o beta testing

7) System implementation: - after completing the testing phase we have to implement a particular product or system according to the customer need. In

the implementation phase some design and other user activity part may be changed as per customer need.

8) System maintenance: - after implementation the users use the particular software to their corresponding operation to active there job. In this phase the software maintained from the user or developer side after spanning some times of use of particular software. In this phase the related hardware, software and other utilities are also maintained.

9) System documentation: - documentation refers the approach and guidelines for the user as well as the customer to the related software. The documentation refers some writing instruction for how to use for related hardware requirement, and also some maintains factors for the users.

TOPIC 2
SOFTWARE PROCESS MODELS

A software process is a coherent set of activities for specifying, designing, implementing and testing software systems. A structured set of activities required to develop a software system:

- Specification

- Design

- Validation

- Evolution

A software process model is an abstract representation of a process. It presents a description of a process from some particular perspective.

Generic software process models

a) The waterfall model - Separate and distinct phases of specification and development.

b) Evolutionary development - Specification and development are interleaved.

c) Formal systems development - A mathematical system model is formally transformed to an implementation.

d) Reuse-based development - The system is assembled from existing components

- ***Linear/waterfall model***

This model assumes that everything is carried out and taken place perfectly as planned in the previous stage and there is no need to think about the past issues that may arise in the next phase. This model does not work smoothly if there are some issues left at the previous step. The sequential nature of model does not allow us go back and undo or redo our actions.

This model is best suited when developers already have designed and developed similar software in the past and is aware of all its domains. The drawbacks of the waterfall model are:

- The difficulty of accommodating change after the process is underway.

- Inflexible partitioning of the project into distinct stages.

- This makes it difficult to respond to changing customer requirements.

- Therefore, this model is only appropriate when the requirements are well-understood.

- *Rapid prototyping*

Rapid Prototyping (RP) can be defined as a group of techniques used to quickly fabricate a scale model of a part or assembly using three-dimensional computer aided design (CAD) data. What is commonly considered to be the first RP technique, Stereo lithography, was developed by 3D Systems of Valencia, CA, USA. The company was founded in 1986, and since then, a number of different RP techniques have become available.

Rapid Prototyping has also been referred to as solid free-form manufacturing; computer automated manufacturing, and layered manufacturing. RP has obvious use as a vehicle for visualization. In addition, RP models can be used for testing, such as when an airfoil shape is put into a wind tunnel. RP models can be used to create male models for tooling, such as silicone rubber molds and investment casts. In some cases, the RP part can be the final part, but typically the RP material is not strong or accurate enough. When the RP material is suitable, highly convoluted shapes (including parts nested within parts) can be produced because of the nature of RP.

The reasons of Rapid Prototyping are

- To increase effective communication.

- To decrease development time.
- To decrease costly mistakes.
- To minimize sustaining engineering changes.
- To extend product lifetime by adding necessary features and eliminating redundant features early in the design.

Rapid Prototyping decreases development time by allowing corrections to a product to be made early in the process. By giving engineering, manufacturing, marketing, and purchasing a look at the product early in the design process, mistakes can be corrected and changes can be made while they are still inexpensive. The trends in manufacturing industries continue to emphasize the following:

- Increasing number of variants of products.

- Increasing product complexity.

- Decreasing product lifetime before obsolescence.

- Decreasing delivery time.

Rapid Prototyping improves product development by enabling better communication in a concurrent engineering environment.

Methodology of Rapid Prototyping

The basic methodology for all current rapid prototyping techniques can be summarized as follows:

1. A CAD model is constructed, and then converted to STL format. The resolution can be set to minimize stair stepping.
2. The RP machine processes the .STL file by creating sliced layers of the model.
3. The first layer of the physical model is created. The model is then lowered by the thickness of the next layer, and the process is repeated until completion of the model.
4. The model and any supports are removed. The surface of the model is then finished and cleaned.

- ***Evolutionary models***

This approach is based on the idea of rapidly developing an initial software implementation from very abstract specifications and modifying this according to your appraisal. Each program version inherits the best features from earlier versions. Each version is refined based upon feedback from yourself to produce a system which satisfies your needs. At this point the system may be delivered or it may be re-implemented using a more structured approach to enhance robustness and maintainability. Specification, development and validation activities are concurrent with strong feedback between each.
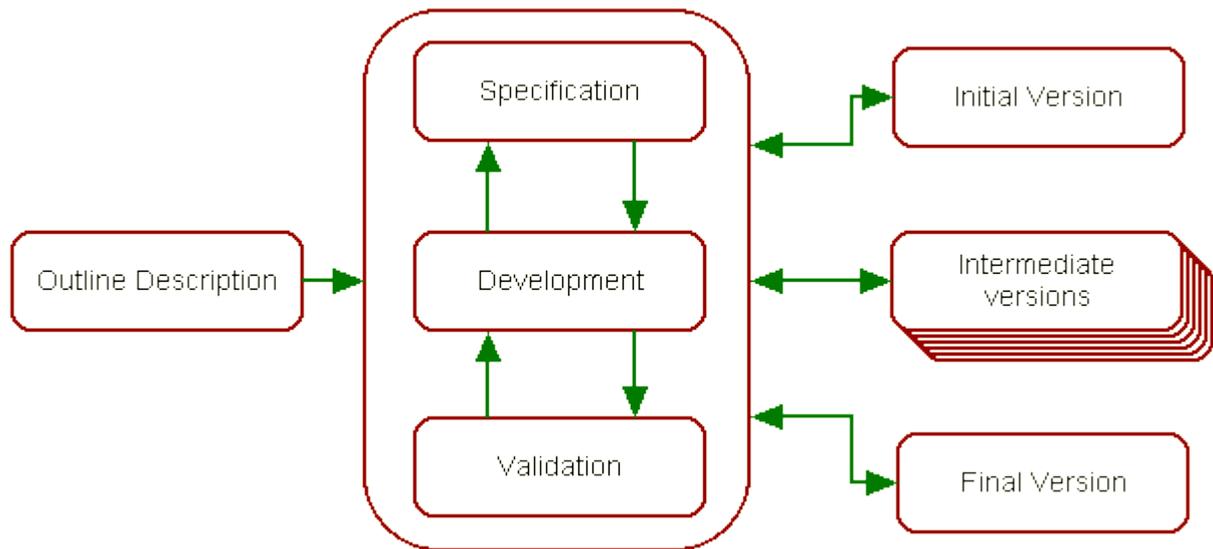
There are two types of evolutionary development:

1. **Exploratory programming**
   here, the objective of the process is to work with you to explore their requirements and deliver a final system. The development starts with better understood components of the system. The software evolves by adding new features as they are proposed.

2. **Throwaway prototyping**
   here, the purpose of the evolutionary development process is to understand your requirements and thus develop a better requirements definition for the system. The prototype concentrates on experimenting with those components of the requirements which are poorly understood.

Advantages
This is the only method appropriate for situations where a detailed system specification is unavailable. Effective in rapidly producing small systems, software with short life spans, and developing sub-components of larger systems.

Disadvantages
It is difficult to measure progress and produce documentation reflecting every version of the system as it evolves. This paradigm usually results in badly structured programs due to continual code modification. Production of good quality software using this method requires highly skilled and motivated programmers.

Problems of evolutionary development include:

- Lack of process visibility.

- Systems are often poorly structured.

- Special skills (e.g. in languages for rapid prototyping) may be required.

Evolutionary development is usually employed in:

- For small or medium-size interactive systems

- For parts of large systems (e.g. the user interface)

- For short-lifetime systems

- ### *Component based models*

It's the creation, integration, and re-use of components of program code, each of which has a common interface for use by multiple systems.
Based on systematic reuse where systems are integrated from existing components or COTS (Commercial-off-the-shelf) systems. Process stages include:

- Component analysis

- Requirements modification

- System design with reuse

- Development and integration

This approach is becoming more important but still limited experience with it.
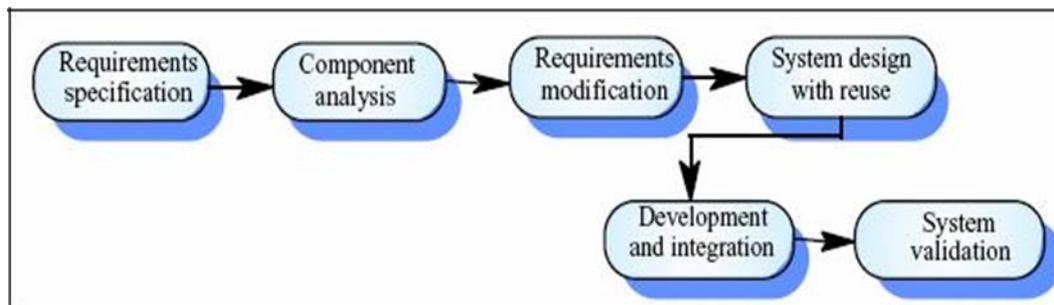
- ### *Other models*

Process Iteration

System requirements ALWAYS evolve in the course of a project so process iteration where earlier stages are reworked is always part of the process for large systems. Iteration can be applied to any of the generic process models. Two (related) approaches:

- Incremental development

- Spiral development

Incremental Development

Rather than deliver the system as a single delivery, the development and delivery is broken down into increments with each increment delivering part of the required

functionality. User requirements are prioritized and the highest priority requirements are included in early increments. Once the development of an increment is started, the requirements are frozen though requirements for later increments can continue to evolve.

Advantages of the incremental development include:

- Customer value can be delivered with each increment so system functionality is available earlier.

- Early increments act as a prototype to help elicit requirements for later increments.

- Lower risk of overall project failure.

The highest priority system services tend to receive the most testing.
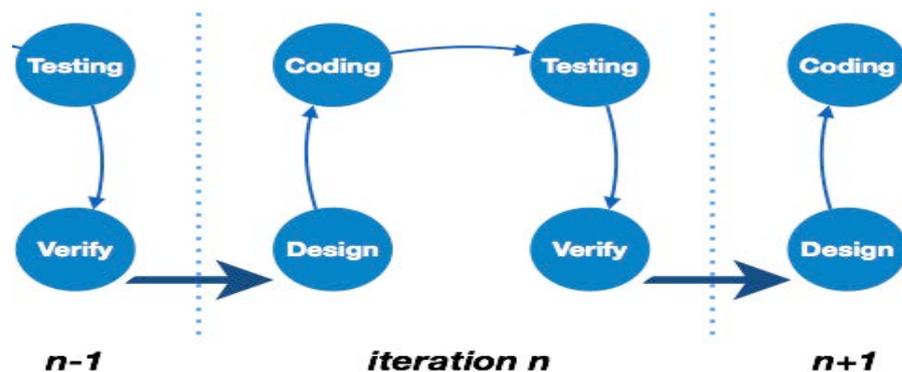
Spiral Development

Process is represented as a spiral rather than as a sequence of activities with backtracking. Each loop in the spiral represents a phase in the process. No fixed phases such as specification or design loops in the spiral are chosen depending on what is required. Risks are explicitly assessed and resolved throughout the process.

The spiral model sectors include:

- Objective setting - Specific objectives for the phase are identified.

- Risk assessment and reduction - Risks are assessed and activities put in place to reduce the key risks.

- Development and validation - A development model for the system is chosen which can be any of the generic models

- Planning - The project is reviewed and the next phase of the spiral is planned.

CASE

Computer-aided software engineering (CASE) is software to support software development and evolution processes. Activity automation includes:

- Graphical editors for system model development

- Data dictionary to manage design entities

- Graphical UI builder for user interface construction

- Debuggers to support program fault finding

- Automated translators to generate new versions of a program

Case Technology

Case technology has led to significant improvements in the software process though not the order of magnitude improvements that were once predicted. Reasons include:

- Software engineering requires creative thought - this is not readily automatable.

- Software engineering is a team activity and, for large projects, much time is spent in team interactions. CASE technology does not really support these.

CASE Classification

Classification helps us understand the different types of CASE tools and their support for process activities.

- Functional perspective - Tools are classified according to their specific function.

- Process perspective - Tools are classified according to process activities that are supported.

- Integration perspective - Tools are classified according to their organization into integrated units.

Functional Tool Classification	
Tool Type	**Examples**
Planning tools	PERT tools, estimation tools, spreadsheets
Editing tools	Text editors, diagram editors, word processors
Change management tools	Requirements traceability tools, change control systems
Configuration management tools	Version management systems, system building tools
Prototyping tools	Very high-level languages, user interface generators
Method-support tools	Design editors, data dictionaries, code generators
Language-processing tools	Compilers, interpreters
Program analysis tools	Cross reference generators, static analyzers, dynamic analyzers
Testing tools	Test data generators, file comparators
Debugging tools	Interactive debugging systems
Documentation tools	Page layout programs, image editors
Re-engineering tools	Cross-reference systems, program restructuring systems

CASE Integration

Tools - Support individual process tasks such as design consistency checking, text editing, etc.

Workbenches - Support a process phase such as specification or design. Normally include a number of integrated tools Environments - Support all or a substantial part of an entire software process. Normally include several integrated workbenches
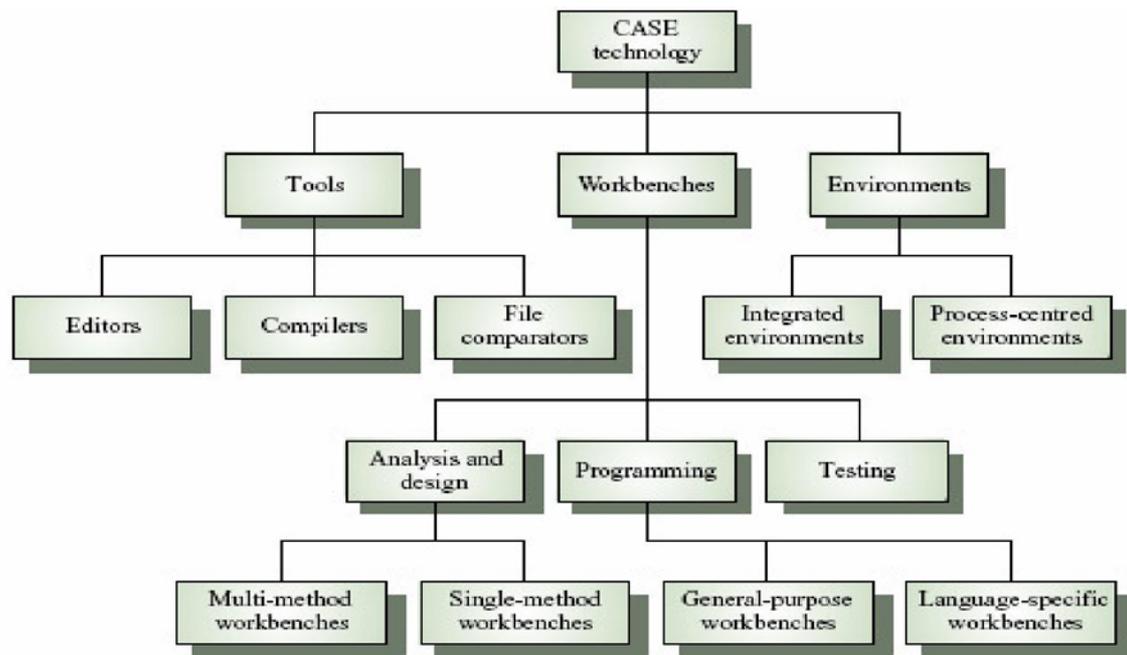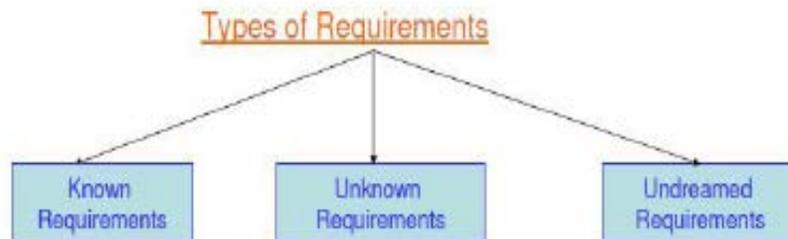
TOPIC 3
SOFTWARE REQUIREMENTS ANALYSIS

- ## *Overview or requirements concepts*

Software Requirement

1. A condition of capability needed by a user to solve a problem or achieve an objective
2. A condition or a capability that must be met or possessed by a system to satisfy a contract, standard, specification, or other formally imposed document."

1   **Known Requirements: -** Something a stakeholder believes to be implemented
2   **Unknown requirements:-** Forgotten by the stakeholder because they are not needed right now or needed only by another stakeholder
3   **Undreamt requirements:-** stakeholder may not be able to think of new requirement due to limited knowledge

A Known, Unknown and Undreamt requirements may be functional or non-functional.

- **Functional requirements: -** describe what the software has to do. They are often called product features. It depends on the type of software, expected users and the type of system where the software is used.
- **Non Functional requirements**: - are mostly quality requirements. That stipulate how well the software does, what it has to do. These define system properties and constraints e.g. reliability, response time and storage requirements. Constraints are I/O device capability, system representations, etc.
- **User requirements: -** Statements in natural language plus diagrams of the services the system provides and its operational constraints.
- **System requirements: -** A structured document setting out detailed descriptions of the system's functions, services and operational constraints. Defines what should be implemented so may be part of a contract between client and contractor.

Requirements engineering Process

• The process of finding out, analyzing, documenting and checking these services and constraints called requirement engineering.
• RE produces one large document, written in a natural language, contains a description of what the system will do without how it will do
• Input to RE is problem statement prepared by the customer and the output is SRS prepared by the developer.
• The requirements themselves are the descriptions of the system services and constraints that are generated during the requirements engineering process

Definition 2

Requirements engineering processes: - The processes used for RE vary widely depending on the application domain, the people involved and the organisation developing the requirements. However, there are a number of generic activities common to all processes.

- Requirements elicitation
- Requirements analysis
- Requirements documentation
- Requirements review

Crucial process steps of Requirement Engineering
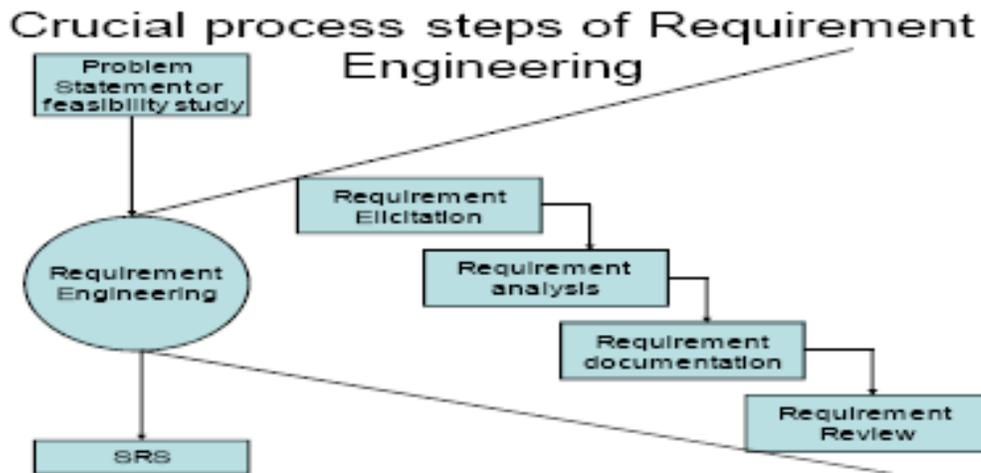
- **Requirement Elicitation: -** known as gathering of requirement. Here requirement are identified with the help of customer and exiting system processes, if they are available.
- **Requirement Analysis: -** analysis of requirement starts with Requirement Elicitation. Requirements are analyzed in order to identify inconstancy, defects etc.
- **Requirement Documentation: -** this is the end product of requirement elicitation and analysis. Documentation is very important as it will be the foundation for the design of the software. The documentation is known as SRS.
- **Requirement Review:-**review process is carried out to improve the quality of the SRS. It may also be called verification. It should be a continuous activity that is incorporated into the elicitation, analysis, documentation.

The primary output of requirement engineering process is system requirement specification (SRS).

Feasibility Study: - It is the process of evaluation or analysis of the potential impact of a propose project or program.

Five common factors (TELOS)

- ❖ **Technical feasibility:** - Is it technically feasible to provide direct communication connectivity through space from one location of globe to another location?
- ❖ **Economic feasibility:-**Are the project's cost assumption realistic?
- ❖ **Legal feasibility:** - Does the business model realistic?
- ❖ **Operational feasibility: -** Is there any market for the product?
- ❖ **Schedule feasibility: -** Are the project's schedule assumption realistic?

Elicitation: - It is also called requirement discovery. Requirements are identified with the help of customer and exiting system processes, if they are available.
Requirement Elicitation is most difficult, perhaps most critical, most error prone and most communication intensive aspect of software development.

Various methods of Requirements Elicitation

1. Interviews
• After receiving the problem statement from the customer, the first step is to arrange a meeting with the customer.
• During the meeting or interview, both the parties would like to understand each other.
• The objective of conducting an interview is to understand the customer's expectation from the software

Selection of stakeholder
1. Entry level personnel
2. Middle level stakeholder
3. Managers
4. Users of the software (Most important)

2. Brainstorming Sessions
• Brainstorming is a group technique that may be used during requirement elicitation to understand the requirement
• Every idea will be documented in a way that every can see it
• After brainstorming session a detailed report will be prepared and reviewed by the facilitator.

3. Facilitated Application specification approach (FAST)
• FAST is similar to brainstorming session and the objective is to bridge the expectation gap-a difference what the developers think they are supposed to build and what the customer think they are going to get
• In order to reduce the gap a team oriented approach is developed for requirement gathering and is called FAST.
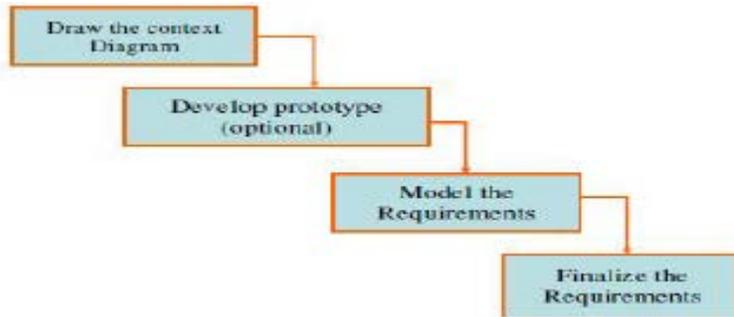
4. Quality function deployment (QFT)
It incorporates the voice of the customer and converts it in to the document.

- *Requirement analysis process*

Requirement analysis phase analyze, refine and scrutinize requirements to make consistent & unambiguous requirements

1   **Draw the context diagram**
    The context diagram is a simple model that defines the boundaries and interface of the
    proposed system.
2   **Development of prototype**
    Prototype helps the client to visualize the proposed system and increase the understanding of
    requirement. Prototype may help the parties to take final decision.
3   **Model the requirement**
    This process usually consists of various graphical representations of function, data entities,
    external entities and the relationship between them. It graphical view may help to find
    incorrect, inconsistent, missing requirements. Such models include data flow diagram, entity
    relationship diagram, data dictionary, state transition diagram.
4   **Finalize the requirement**
    After modeling the requirement inconsistencies and ambiguities have been identified and
    corrected. Flow of data among various modules has been analyzed. Now Finalize and
    analyzes requirements and next step is to document these requirements in prescribed format.

Documentation
This is the way of representing requirements in a consistent format SRS serves many purpose
depending upon who is writing it.

- *Requirements specification*

Requirements specification is a complete description of the behavior of a system to be
developed. It includes a set of use cases that describe all the interactions the users will have with
the software. Use cases are also known as functional requirements. In addition to use cases, the
SRS also contains non-functional (or supplementary) requirements. Nonfunctional requirements
are requirements which impose constraints on the design or implementation (such as
performance engineering requirements, quality standards, or design constraints)

Need for Software Requirement Specification (SRS)
- The problem is that the client usually does not understand software or the software
  development process, and the developer often does not understand the client's problem
  and application area
- This causes a communication gap between the parties involved in the development
  project. A basic purpose of software requirements specification is to bridge this
 communication gap.

Characteristics of good SRS document: - Some of the identified desirable qualities of the SRS documents are the following-

- **Concise-** The SRS document should be concise and at the same time unambiguous, consistent, and complete. An SRS is unambiguous if and only if every requirement stated has one and only one interpretation.
- **Structured-** The SRS document should be well-structured. A well-structured document is easy to understand and modify. In practice, the SRS document undergoes several revisions to cope with the customer requirements.
- **Black-box view-** It should only specify what the system should do and refrain from stating how to do. This means that the SRS document should specify the external behaviours of the system and not discuss the implementation issues.
- **Conceptual integrity-** The SRS document should exhibit conceptual integrity so that the reader can easily understand the contents.
- **Verifiable-** All requirements of the system as documented in the SRs document should be verifiable. This means that it should be possible to determine whether or not requirements have been met in an implementation.
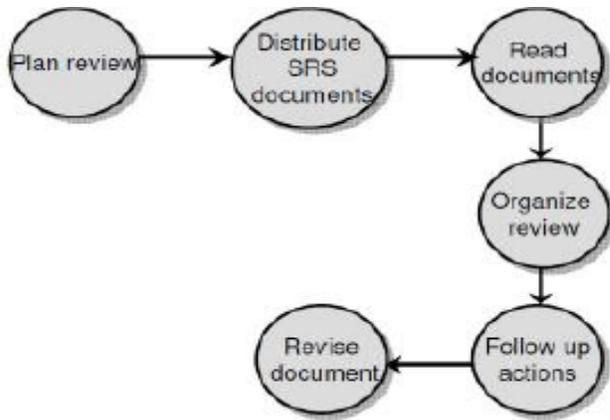
Requirements Validation
Requirement Validation is used for checking the document:-

- Completeness & consistency
- Conformance to standards
- Requirements conflicts
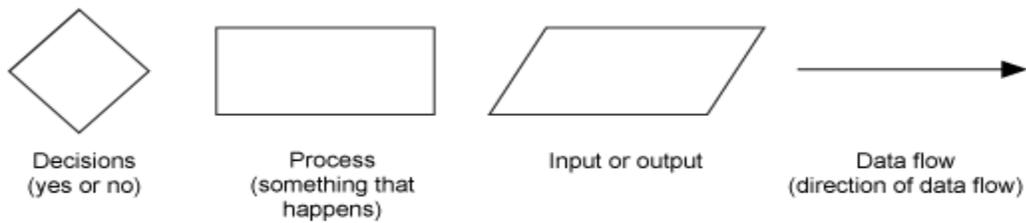- Technical errors
- Ambiguous requirements
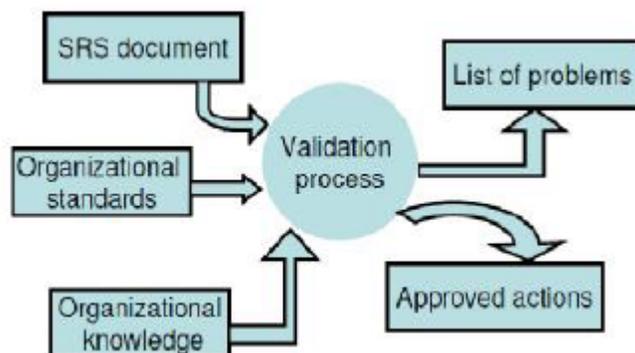
TOPIC 4
DESIGN TOOLS AND METHODS

- ## *System flowcharts*

System flowcharts are a way of displaying how data flows in a system and how decisions are made to control events.

To illustrate this, symbols are used. They are connected together to show what happens to data and where it goes. The basic ones include:

Decisions (yes or no) | Process (something that happens) | Input or output | Data flow (direction of data flow)

Symbols used in flow charts

Note that system flow charts are very similar to data flow charts. Data flow charts do not include decisions, they just show the path that data takes, where it is held, processed, and then output.

Using system flowchart ideas

This system flowchart is a diagram for a 'cruise control' for a car. The cruise control keeps the car at a steady speed that has been set by the driver.
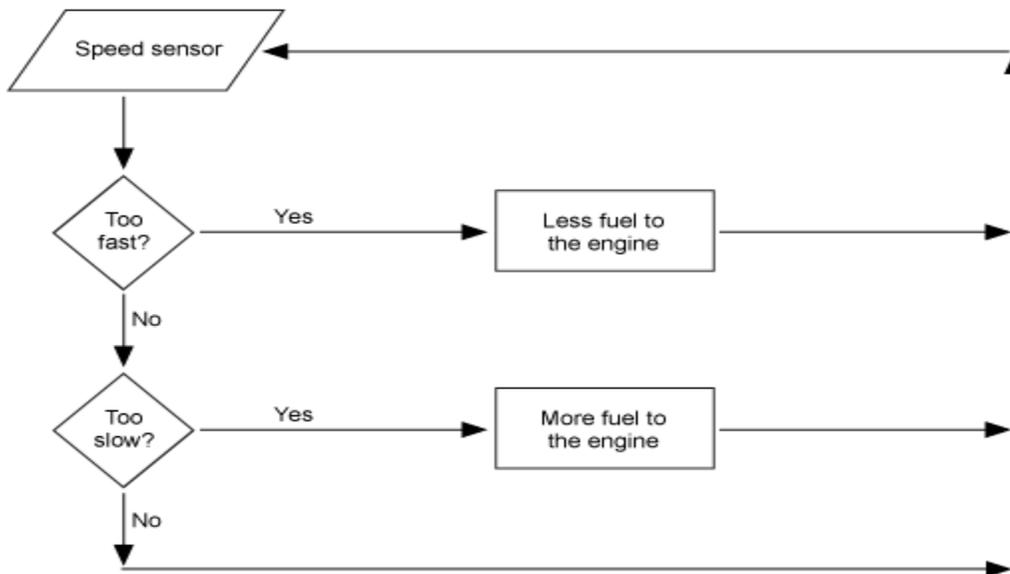
A system flowchart for cruise control on a car

The flowchart shows what the outcome is if the car is going too fast or too slow. The system is designed to add fuel, or take it away and so keep the car's speed constant. The output (the car's new speed) is then fed back into the system via the speed sensor.

Other examples of uses for system diagrams include:

- aircraft control
- central heating
- automatic washing machines
- booking systems for airlines

Types of flowchart

Sterneckert (2003) suggested that flowcharts can be modeled from the perspective of different user groups (such as managers, system analysts and clerks) and that there are four general types:

- *Document flowcharts*, showing controls over a document-flow through a system
- *Data flowcharts*, showing controls over a data-flow in a system
- *System flowcharts* showing controls at a physical or resource level
- *Program flowchart*, showing the controls in a program within a system

Notice that every type of flowchart focuses on some kind of control, rather than on the particular flow itself.
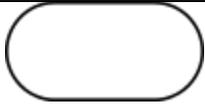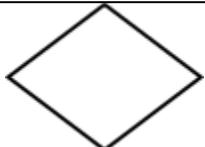
However, there are several of these classifications. For example, Andrew Veronis (1978) named three basic types of flowcharts: the *system flowchart*, the *general flowchart*, and the *detailed flowchart*. That same year Marilyn Bohl (1978) stated "in practice, two kinds of flowcharts are used in solution planning: *system flowcharts* and *program flowcharts*. More recently Mark A. Fryman (2001) stated that there are more differences: "Decision flowcharts, logic flowcharts, systems flowcharts, product flowcharts, and process flowcharts are just a few of the different types of flowcharts that are used in business and government".

In addition, many diagram techniques exist that are similar to flowcharts but carry a different name, such as UML activity diagrams.

Flowchart building blocks

Common Shapes

The following are some of the commonly used shapes used in flowcharts. Generally, flowcharts flow from top to bottom and left to right.

Shape	Name	Description
	Flow Line	An arrow coming from one symbol and ending at another symbol represents that control passes to the symbol the arrow points to. The line for the arrow can be solid or dashed. The meaning of the arrow with dashed line may differ from one flowchart to another and can be defined in the legend.
	On-Page Connector	Generally represented with a circle, showing where multiple control flows converge in a single exit flow. It will have more than one arrow coming into it, but only one going out. In simple cases, one may simply have an arrow point to another arrow instead. These are useful to represent an iterative process (what in Computer Science is called a loop). A loop may, for example, consist of a connector where control first enters, processing steps, a conditional with one arrow exiting the loop, and one going back to the connector. For additional clarity, wherever two lines accidentally cross in the drawing, one of them may be drawn with a small semicircle over the other, showing that no connection is intended.
	Annotation	Annotations represent comments or remarks about the flowchart. Like comments found in high-level programming languages, they have no effect on the interpretation or behavior of the flowchart. Sometimes, the shape consists of a box with dashed (or dotted) lines.
	Terminal	Represented as circles, ovals, stadiums or rounded (fillet) rectangles. They usually contain the word "Start" or "End", or another phrase signaling the start or end of a process, such as "submit inquiry" or "receive product".
	Decision	Represented as a diamond (rhombus) showing where a decision is necessary, commonly a Yes/No question or True/False test. The conditional symbol is peculiar in that it has two arrows coming out of it, usually from the bottom point and right point, one corresponding to Yes or True, and one corresponding to No or False. (The arrows should always be labeled.) More than two arrows can be used, but this is normally a clear indicator that a complex decision is being taken, in which case it may need to be broken-down further or replaced with the "predefined process" symbol. Decision can also help in the filtering of data.
	Input/output	Represented as a parallelogram. Involves receiving data and displaying processed data. Can only move from input to output and not vice versa. Examples: Get X from the user; display X.
	Predefined Process	Represented as rectangles with double-struck vertical edges; these are used to show complex processing steps which may be detailed in a separate flowchart. Example: PROCESS-FILES. One subroutine may have multiple distinct entry points or exit flows (see coroutine). If so, these are shown as labeled 'wells'

		in the rectangle, and control arrows connect to these 'wells'.
	Process	Represented as rectangles. This shape is used to show that something is performed. Examples: "Add 1 to X", "replace identified part", "save changes", etc....
	Preparation	Represented as a hexagon. May also be called initialization. Shows operations which have no effect other than preparing a value for a subsequent conditional or decision step. Alternatively, this shape is used to replace the Decision Shape in the case of conditional looping.
	Off-Page Connector	Represented as a home plate-shaped pentagon. Similar to the on-page connector except allows for placing a connector that connects to another page.

Other Shapes

A typical flowchart from older basic computer science textbooks may have the following kinds of symbols:

Labeled connectors

Represented by an identifying label inside a circle. Labeled connectors are used in complex or multi-sheet diagrams to substitute for arrows. For each label, the "outflow" connector must always be unique, but there may be any number of "inflow" connectors. In this case, a junction in control flow is implied.

Concurrency symbol

Represented by a double transverse line with any number of entry and exit arrows. These symbols are used whenever two or more control flows must operate simultaneously. The exit flows are activated concurrently, when all of the entry flows have reached the concurrency symbol. A concurrency symbol with a single entry flow is a *fork*; one with a single exit flow is a *join*.

Data-flow extensions

A number of symbols have been standardized for data flow diagrams to represent data flow, rather than control flow. These symbols may also be used in control flowcharts (e.g. to substitute for the parallelogram symbol).

- A *Document* represented as a rectangle with a wavy base;
- A *Manual input* represented by quadrilateral, with the top irregularly sloping up from left to right. An example would be to signify data-entry from a form;
- A *Manual operation* represented by a trapezoid with the longest parallel side at the top, to represent an operation or adjustment to process that can only be made manually.
- A *Data File* represented by a cylinder.

- *Case tools*

Computer-aided software engineering (**CASE**) is the domain of software tools used to design and implement applications. CASE tools are similar to and were partly inspired by Computer Aided Design (CAD) tools used to design hardware products. CASE tools are used to develop software that is high-quality, defect-free, and maintainable. CASE software is often associated with methodologies for the development of information systems together with automated tools that can be used in the software development process.

Case software

CASE software is classified into 3 categories:

1. *Tools* support specific tasks in the software life-cycle.
2. *Workbenches* combine two or more tools focused on a specific part of the software life-cycle.
3. *Environments* combine two or more tools or workbenches and support the complete software life-cycle.

Tools

CASE tools support specific tasks in the software development life-cycle. They can be divided into the following categories:

1. Business and Analysis modeling. Graphical modeling tools. E.g., E/R modeling, object modeling, etc.
2. Development. Design and construction phases of the life-cycle. Debugging environments. E.g., GNU Debugger.
3. Verification and validation. Analyze code and specifications for correctness, performance, etc.
4. Configuration management. Control the check-in and check-out of repository objects and files. E.g., SCCS, CMS.
5. Metrics and measurement. Analyze code for complexity, modularity (e.g., no "go to's"), performance, etc.
6. Project management. Manage project plans, task assignments, scheduling.

Another common way to distinguish CASE tools is the distinction between Upper CASE and Lower CASE. Upper CASE Tools support business and analysis modeling. They support traditional diagrammatic languages such as ER diagrams, Data flow diagram, Structure charts, Decision Trees, Decision tables, etc. Lower CASE Tools support development activities, such as physical design, debugging, construction, testing, component integration, maintenance, and reverse engineering. All other activities span the entire life-cycle and apply equally to upper and lower CASE.

Workbenches

Workbenches integrate two or more CASE tools and support specific software-process activities. Hence they achieve:

- A homogeneous and consistent interface (presentation integration).
- Seamless integration of tools and tool chains (control and data integration).

An example workbench is Microsoft's Visual Basic programming environment. It incorporates several development tools: a GUI builder, smart code editor, debugger, etc. Most commercial CASE products tended to be such workbenches that seamlessly integrated two or more tools. Workbenches also can be classified in the same manner as tools; as focusing on Analysis, Development, Verification, etc. as well as being focused on upper case, lower case, or processes such as configuration management that span the complete life-cycle.

Environments

An environment is a collection of CASE tools or workbenches that attempts to support the complete software process. This contrasts with tools that focus on one specific task or a specific part of the life-cycle. CASE environments are classified as follows:

1. Toolkits. Loosely coupled collections of tools. These typically build on operating system workbenches such as the Unix Programmer's Workbench or the VMS VAX set. They typically perform integration via piping or some other basic mechanism to share data and pass control. The strength of easy integration is also one of the drawbacks. Simple passing of parameters via technologies such as shell scripting can't provide the kind of sophisticated integration that a common repository database can.
2. Fourth generation. These environments are also known as 4GL standing for fourth generation language environments due to the fact that the early environments were designed around specific languages such as Visual Basic. They were the first environments to provide deep integration of multiple tools. Typically these environments were focused on specific types of applications. For example, user-interface driven applications that did standard atomic transactions to a relational database. Examples are Informix 4GL, and Focus.
3. Language-centered. Environments based on a single often object-oriented language such as the Symbolics Lisp Genera environment or Visual Works Smalltalk from Parcplace. In these environments all the operating system resources were objects in the object-oriented language. This provides powerful debugging and graphical opportunities but the code developed is mostly limited to the specific language. For this reason, these environments were mostly a niche within CASE. Their use was mostly for prototyping and R&D projects. A common core idea for these environments was the model-view-controller user interface that facilitated keeping multiple presentations of the same design consistent with the underlying model. The MVC architecture was adopted by the other types of CASE environments as well as many of the applications that were built with them.
4. Integrated. These environments are an example of what most IT people tend to think of first when they think of CASE. Environments such as IBM's AD/Cycle, Andersen Consulting's FOUNDATION, the ICL CADES system, and DEC Cohesion. These environments attempt to cover the complete life-cycle from analysis to maintenance and provide an integrated database repository for storing all artifacts of the software process. The integrated software repository was the defining feature for these kinds of tools. They provided multiple different design models as well as support for code in heterogeneous