SOFTWARE ENGINEERING

(CSE3005)

7/18/2019

Prof. Anand Motwani
Faculty, SCSE
VIT Bhopal University

Unit I

SOFTWARE PROCESS and MODELS

The Nature Of Software, Software Engineering, Software Process - Software Myths - Process Models – Generic –Perspective – Specialized – The Unified Process – Personal And Team Software Process - Agile Development – Agile Process- Extreme Programming - Software Engineering Knowledge – Core Principles.

Aims and Objectives

- To introduce the basic concepts of software development Process, models and methods.
- To appraise the need for a professional approach to software system development;

1. Introduction

Computer software has become a driving force. It is the engine that drives business decision making. It serves as the basis for modern scientific investigation and engineering problem solving. It is a key factor that differentiates modern products and services. . It is embedded in systems of all kinds: transportation, medical, telecommunications, military, industrial processes, entertainment, office products... Software is virtually inescapable in a modern world. And as we move into the 21st century, it will become the driver for new advances in everything from elementary education to genetic engineering.

Software's impact on our society and culture continues to be profound. As its importance grows, the software community continually attempts to develop technologies that will make it easier, faster, and less expensive to build high-quality computer programs. Some of these technologies are targeted at a specific application domain (e.g., Web-site design and implementation); others focus on a technology domain (e.g., object-oriented systems); and still others are broad-based (e.g., operating systems such as LINUX).

Prepared by:
Prof. Anand Motwani
Faculty SCSE,
VIT Bhopal University

2. What is software? Write it's roles.

Software

Not only the computer program (s) but also

1) Instructions (programs) that when executed provide desired function and performance

2) Data structures that enable the programs to adequately manipulate information

3) Documents that describe the operation and use of the programs

So, Software is a set of items or objects that form a "configuration" that includes programs, data structures, documents and data etc.

Or

Alternate definition: Software is the collection of computer programs, procedures, Rules and associate with documentation and data. Product that engineers design & build

(1) Programs that execute within a computer of any size & architecture.

(2) Documents

(3) Data (Text, Video etc.)

Software is everywhere: PC, phones, electronic devices (e.g. TV, washing machine, . . .), computing centre, Web server, . . .

Software's Dual Roles

- Software is a product
  - Delivers computing potential Delivers potential
  - Produces, manages, acquires, modifies, displays, or transmits information
- Software is a vehicle for delivering a product Software product
  - Supports or directly provides system functionality Supports functionality
  - Controls other programs (e.g., an operating system)
  - Effects communications (e.g., networking software)
  - Helps build other software (e.g., software tools)

Prepared by:
Prof. Anand Motwani
Faculty SCSE,
VIT Bhopal University

3. Define Software Engineering?

Software Engineering is a systematic approach to development, operation, maintenance and retirement of software.

Or

Software Engineering is the application of science and mathematics by which the capabilities of computer equipment are made useful to man via computer programs, procedures and associated with documentation.

Or

Software engineering (SE) is the application of systematic, disciplined, quantifiable approach to the development, operation and maintenance of software; that is, the application of engineering to software.

Or

SE is an engineering discipline concerned with all aspects of software production from early specifications through to maintenance.

Or

SE is the profession that creates and maintains software applications by applying technologies and practices from computer science, project management, engineering, application domain, and other fields.

Goal of the Software Engineering is to produce high quality software at low cost.

Prepared by:
Prof. Anand Motwani
Faculty SCSE,
VIT Bhopal University

4. Write Characteristics of Software. OR Write about nature of Software.

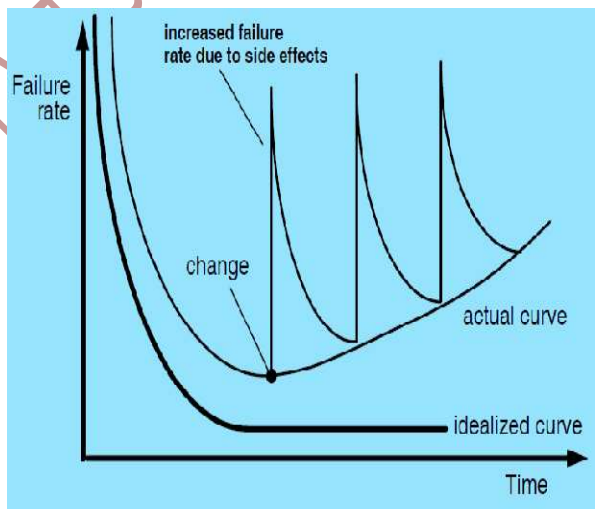
The economies of ALL developed nations are dependent on software. More and more systems are software controlled Software engineering is concerned with theories, methods and tools for professional software development. Expenditure on software represents a significant fraction of GNP in all developed countries.

- **Software is engineered:** Software is engineered or developed, not manufactured in the traditional sense. Some general approaches to develop software.
  - Develop and use good engineering practices for building software.
  - Make heavy use of reusable software components.
  - Use modern languages that support good software development practices, e.g. Java.
  - Use 4th generation languages.

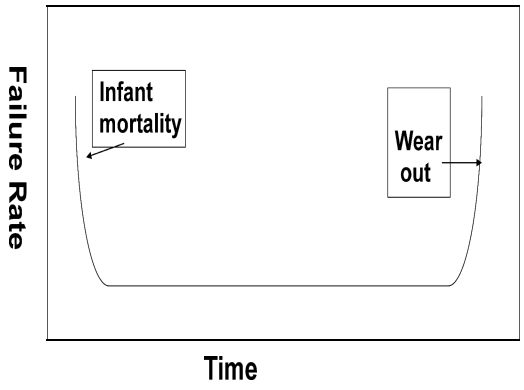
But, almost everything is a two-edged sword while considering long term tool maintenance.

- **Software does not wear out:** Software does not wear out in the same sense as hardware. In theory, software does not wear out at all but, as Hardware upgrades Software also upgrades.
- **Software is complex**
  - adding another button to an alarm clock costs money during manufacturing of each piece
  - the cost is manufacturer's
  - adding a butting to a program may cost something during software development (or not)

Its consequence is more complex software which may be more difficult to use and the user pays



Prepared by:
Prof. Anand Motwani
Faculty SCSE,
VIT Bhopal University



Bathtub curve

5. Types of Software Applications

- System software
- Application software
- Engineering/scientific software
- Embedded software
- Product-line software
- WebApps (Web applications)
- AI software

Software New Categories Software—

- Ubiquitous computing——wireless networks
- Netsourcing——the Web as a computing engine
- Open source——""free"" source code open to the computing community (a blessing, but also a potential curse!)
- Data mining
- Grid computing
- Cognitive machines
- Software for nanotechnologies

Define Legacy Software.

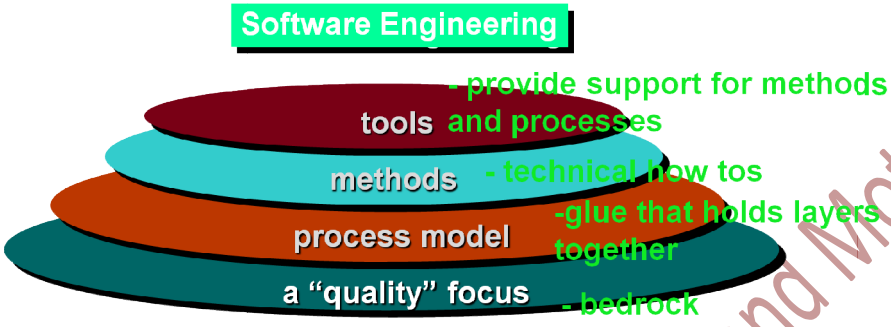
- Software must be adapted to meet the needs of new to computing environments or technology.
- Software must be enhanced to implement new to business requirements.
- Software must be extended to make it interoperable with other more modern systems or databases.
- Software must be re-architected to make it viable within a network environment.

Prepared by:
Prof. Anand Motwani
Faculty SCSE,
VIT Bhopal University

6. Software Engineering is a layered technology. Explain.

Or

Define Layers of Software Engineering.



Software Process

A software process consists of a set of activities and associated results which lead to the production of a software product

Methods

- Includes standards (formal or informal)
- May include conventions, e.g., low level such as naming, variable, language construct use, etc.
- May involve design methodologies.

List the Generic Software Engineering Phases/Activities.

Tools

- Editors
- Design aids
- Compilers
- Computer Aided Software Engineering (CASE)

7. What is a software process?

A set of activities whose goal is the development or evolution of software. Another definition:

It is a series of predictable steps to build a product or system ensuring timely & high quality result.

8. Write Fundamental / Generic activities of all software processes.

Fundamental / Generic activities of all software processes are:

1. Software specification
2. Software design and implementation
3. Software validation
4. Software evolution

Prepared by:
Prof. Anand Motwani
Faculty SCSE,
VIT Bhopal University

Or

System or information engineering (leading to requirements)

- Software project planning
- Requirements analysis
- Development
- Software design
- Coding
- Testing
- Deployment
- Maintenance

Typical activities in these (SE) phases

- Project tracking and control
- Formal reviews
- Software quality assurance
- Configuration management
  - versions, group of versions = configuration
- Documentation
- Reusability management
- Measurements
- Risk management

9. Software Myths:

Definition: Beliefs about software and the process used to build it. Myths have number of attributes that have made them insidious (i.e. dangerous).

- Misleading Attitudes - caused serious problem for managers and technical people.

Management myths

Managers in most disciplines, are often under pressure to maintain budgets, keep schedules on time, and improve quality.

Myth1: We already have a book that's full of standards and procedures for building software, won't that provide my people with everything they need to know?

Reality :

- Are software practitioners aware of existence standards?
- Does it reflect modern software engineering practice?
- Is it complete? Is it streamlined to improve time to delivery while still maintaining a focus on quality?

Myth2*:* If we get behind schedule, we can add more programmers and catch up

Reality*:* Software development is not a mechanistic process like manufacturing. Adding people to a late software project makes it later.

- People can be added but only in a planned and well-coordinated manner

Prepared by:
Prof. Anand Motwani
Faculty SCSE,
VIT Bhopal University

Myth3: If I decide to outsource the software 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 software projects internally, it will invariably struggle when it outsource software projects

Customer Myths

Customer may be a person from inside or outside the company that has requested software under contract.

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

Reality: A poor up-front definition is the major cause of failed software efforts. A formal and detailed description of the information domain, function, behavior, performance, interfaces, design constraints, and validation criteria is essential. These characteristics can be determined only after thorough communication between customer and developer.

Myth: Project requirements continually change, but change can be easily accommodated because software is flexible.

Reality: Customer can review requirements and recommend modifications with relatively little impact on cost. When changes are requested during software design, the cost impact grows rapidly. Below mentioned *figure* for reference.

Practitioner's myths

Myth1: Once we write the program and get it to work, our job is done.

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

Myth2: Until I get the program "running" I have no way of assessing its quality.

Reality: One of the most effective software quality assurance mechanisms can be applied from the inception of a project—the *formal technical review.*

Myth3: The only deliverable work product for a successful project is the working program.

10. Define Process Models (Software Engineering paradigms). How it is chosen?

It is the development strategy that encompasses the process, methods, tools & generic phases i.e. definition, development & support & followed by software engineer or team of E's to develop software.

A simplified representation of a software process, presented from a specific perspective. Examples of process perspectives are

- Workflow perspective - sequence of activities;
- Data-flow perspective - information flow;
- Role/action perspective - who does what.

Few Generic process models are:

- Waterfall;
- Iterative development;

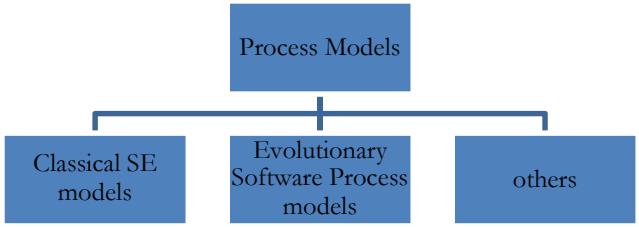
Prepared by:
Prof. Anand Motwani
Faculty SCSE,
VIT Bhopal University

- Component-based software engineering.

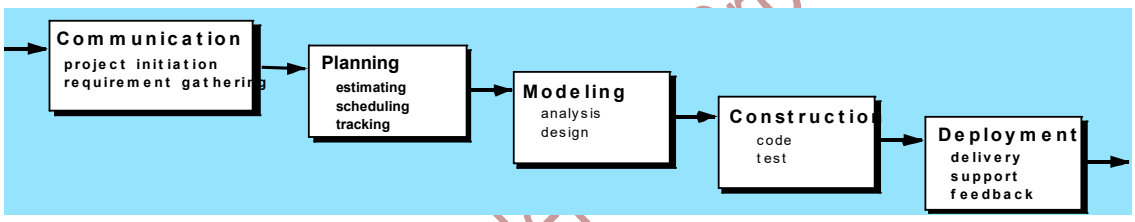
Process models chosen based on:

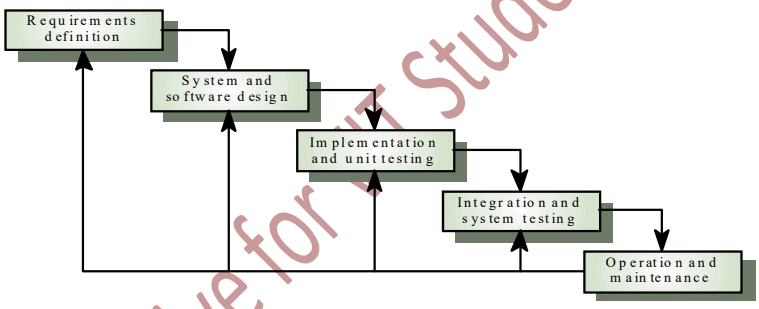
- Nature of project & application
- Methods & Tools to be used
- Controls & deliverables those are required.

11. Show classification of Process Models. Describe few models.



11.1 The Waterfall Model

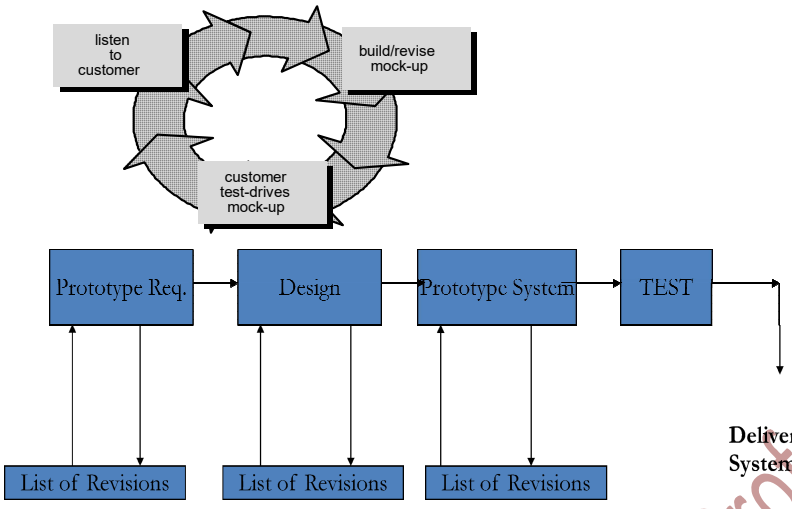




- Requirements analysis and definition
- System and software design
- Implementation and unit testing
- Integration and system testing
- Operation and maintenance
- The drawback of the waterfall model is the difficulty of accommodating change after the process is underway
- Inflexible partitioning of 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.

Prepared by:
Prof. Anand Motwani
Faculty SCSE,
VIT Bhopal University

11.2 Prototyping Prototype Model



Advantages

- A partial product is built in initial stages, customer get chance to see it.
- New requirement easily accommodated.
- More secure, comfortable & satisfied.

Disadvantages

- User not know the difference between prototype and actual well engineered system.
- User expects the early delivery.
- If not managed properly iterative process of prototype can continue for long time.
- Poor documentation.

11.3 Rapid Application Development (RAD)

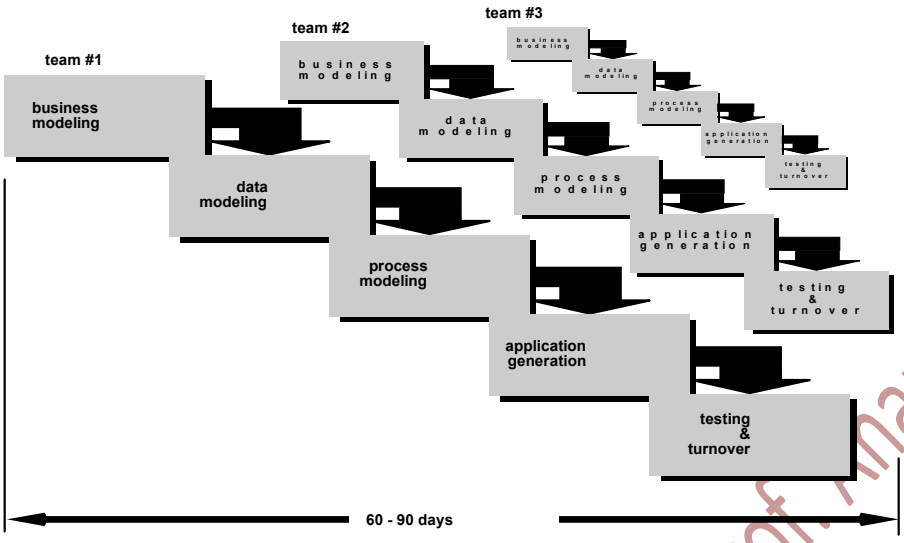
- Proposed by IBM (1980s')
  - High speed adaptation of Linear sequential model
  - Rapid development achieved by using component based construction
  - Increase involvement of customer
  - Rapid prototype delevired to customer (working model)

Advantages

- Customer satisfaction.
- use of tools results into reduced SDLC.

Prepared by:
Prof. Anand Motwani
Faculty SCSE,
VIT Bhopal University

- Reusable components, reduces testing time.
- Feedback available
- Reduced costs



Disadvantages

- For large projects, it required skilled professionals and hence not suitable when technical risks are high (heavy use of new tech.).

- Absence of reusable components, i.e. if high performance good tuning of interfaces required.

11.4 The Incremental Model

- Combines elements of linear seq. model applied repetitively with the iterative philosophy of prototyping.

Incremental development advantages

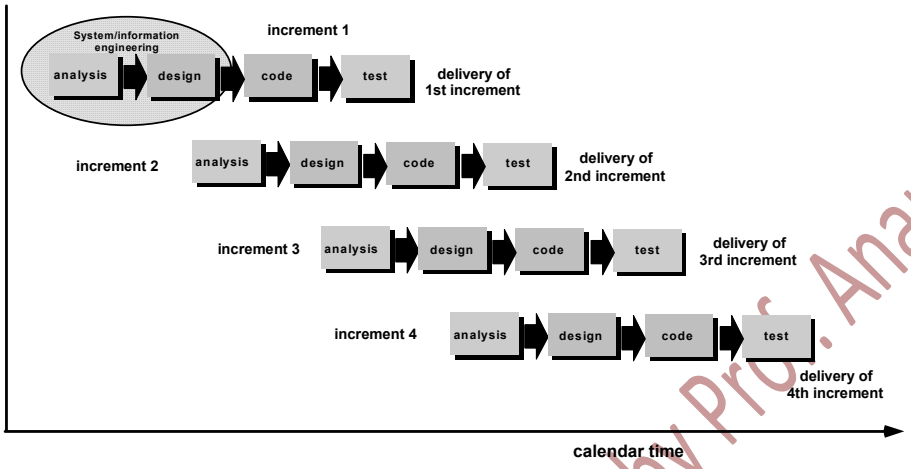
- Customer value can be delivered with each increment so system functionality is available earlier
- Early increments act as a prototype to help elicit requirements for later increments
- Lower risk of overall project failure
- The highest priority system services tend to receive the most testing
- Total cost distributed
- Limited no. of persons
- Development activities for the next release & use of early version done simultaneously
- Testing easy

Prepared by:
Prof. Anand Motwani
Faculty SCSE,
VIT Bhopal University

- Low risk of failures

Incremental development Disadvantages

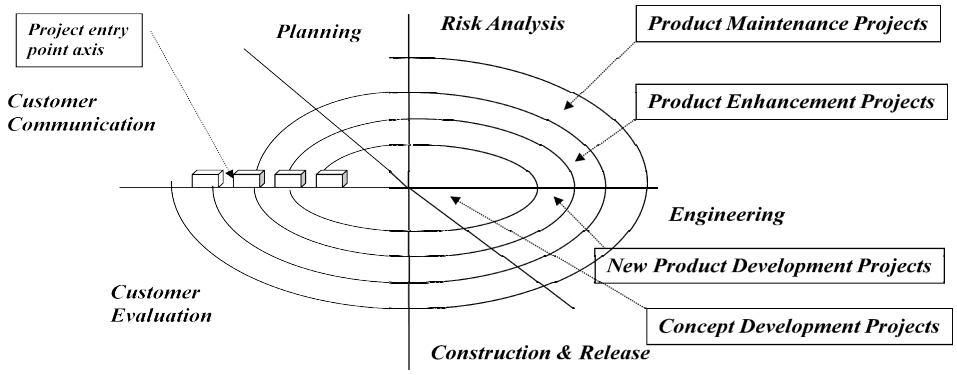
- High Development cost
- Project planning schedule to distribute work
- Testing of modules increases cost.



11.5 An Evolutionary (Spiral) Model

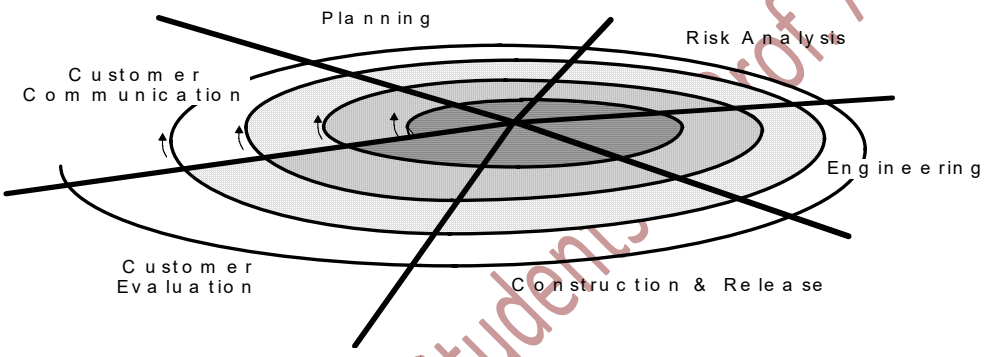
- Proposed by Boehm (1988)
- Combines iterative nature of prototyping with the controlled and systematic approach of Linear Sequential model.
- It provides the potential for rapid development of Incremental versions of software.
- It has 3-6 task regions.
- It explicitly embraces prototyping and an *iterative* approach to software development.
    - Start by developing a small prototype.
    - Followed by a mini-waterfall process, primarily to gather requirements.
    - Then, the first prototype is reviewed.
    - In subsequent loops, the project team performs further requirements, design, implementation and review.
    - The first thing to do before embarking on each new loop is risk analysis.
    - Maintenance is simply a type of on-going development.

Prepared by:
Prof. Anand Motwani
Faculty SCSE,
VIT Bhopal University



The spiral model
-- a realistic approach to the development of large scale systems and software.
- may be difficult to convince customers that the evolutionary approach is controllable.
- a relatively new model, and has not used as widely as the linear sequential or prototyping paradigms.

Phases / Tasks of Evolutionary (Spiral) Model



1. Customer communication 2. Planning 3. Risk Analysis 4. Engineering
5. Construction & Release 6. Customer Evaluation

Each task region is populated by task set.

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.

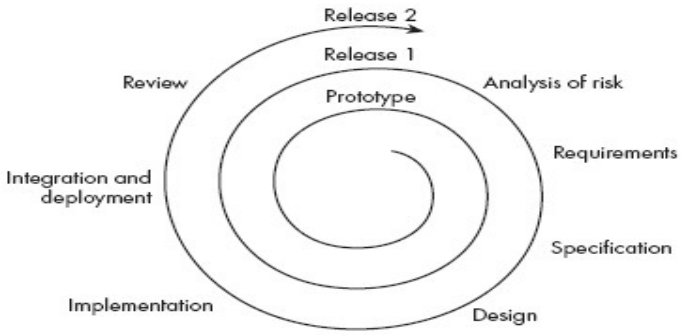
Spiral model sectors

- Objective setting
  - Specific objectives for the phase are identified.
- Risk assessment and reduction

Prepared by:
Prof. Anand Motwani
Faculty SCSE,
VIT Bhopal University

- 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.



Spiral model Advantages

It resolves all possible risks, so get redefined Product. It involves Use of techniques: reuse, prototyping, & component based development which is having its own advantages.

Disadvantages

It is Complex so requires (i) Risk management expertise and (ii) Good management skills.

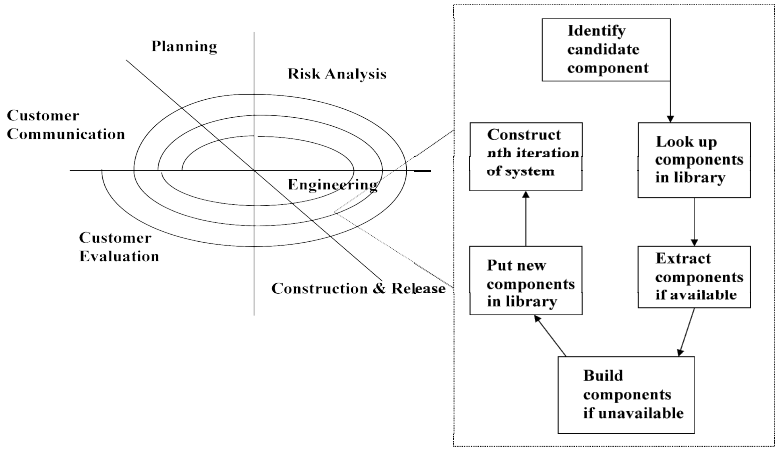
11.6 Component

What is a component?

1. A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of a well-defined architecture. A component conforms to and provides the physical realization of a set of interfaces. (Philippe Krutchen, Rational Software).
2. A runtime software component is a dynamically bindable package of one or more programs managed as a unit and accessed through documented interfaces that can be discovered at runtime. (Gartner Group)
3. A reusable part of software, which is independently developed, and can be brought together with other components to build larger units. It may be adapted but may not be modified. A component can be, for example, a compiled code without a program source, or a part of a model and/or design.

Prepared by:
Prof. Anand Motwani
Faculty SCSE,
VIT Bhopal University

The Component Assembly Model



- Object technologies provide the technical framework for a component-based process model for software engineering.
- The object oriented paradigm emphasizes the creation of classes that encapsulate both data and the algorithm that are used to manipulate the data.
- If properly designed and implemented, object oriented classes are reusable across different applications and computer based system architectures.
- Component Assembly Model leads to software reusability. The integration/assembly of the already existing software components accelerates the development process.
- Nowadays many component libraries are available on the Internet. If the right components are chosen, the integration aspect is made much simpler.
- **Object Technologies -- the technical framework for a component-based process model for software engineering.**
- **The component assembly model incorporates many of the characteristics of the spiral model. It is an evolutionary in nature and demands an iterative approach to the creation of software. So leads to software reuse, and reusability.**
- **Its Advantages includes software reuse, cost reduction and reduction in development cycle time**

Prepared by:
Prof. Anand Motwani
Faculty SCSE,
VIT Bhopal University

12. Rational Unified Process

- Introduction
- Phases
- Core Workflows
- Best Practices
- Tools

Introduction:

The Rational Unified Process™ (RUP) is a Web-enabled software engineering process that enhances team productivity and delivers software best practices to all team members. This e-coach makes process practical by providing prescriptive guidelines, templates and examples for all critical e-development activities. RUP is a customizable framework, which can easily be adapted to work the way you work. It is tightly integrated with Rational tools, allowing development teams to gain the full benefits of the Unified Modeling Language™ (UML), software automation, and other industry best practices.

In short

- A modern process model derived from the work on the UML and associated process.
- It is a good example of hybrid Process model.
- It brings together elements from all of the generic process model, supports iteration, and illustrates good practice in specification & design.
- Normally described from 3 perspectives
    - A dynamic perspective that shows phases over time;
    - A static perspective that shows process activities;
    - A practice perspective that suggests good practices.

- **Team-Unifying Approach**:

The RUP unifies a software team by providing a common view of the development process and a shared vision of a common goal.

- **Increased Team Productivity**
    - Knowledge base of all processes.
    - View of how to develop software
    - Modeling language
    - Rational provides many tools

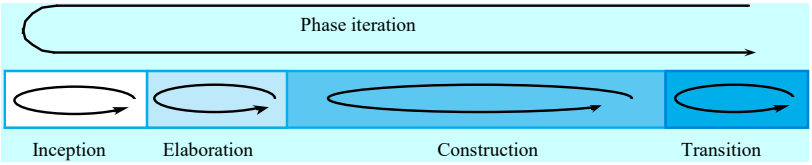
The Rational Unified Process has four phases:

- **Inception** - Define the scope of project. Establish the business case for the system, i.e. identify all external entities (people & systems) that will interact with the system & define these interactions. Then use this information to assess the contribution that the system makes to the business. If this contribution is minor, then the project may be cancelled after this phase.
- **Elaboration** - Plan project, specify features, baseline architecture. Develop an understanding of the problem domain and the system architecture. Develop the project plan & identify key project risks. **After this phase, you should have:** Requirement model for system, Architectural description and Development plan.
- **Construction** - Build the product. System design, programming and testing. Parts of the system are developed in parallel & integrated during this phase. **On**

Prepared by:
Prof. Anand Motwani
Faculty SCSE,
VIT Bhopal University

completion of this phase, you should have: working software system and associated documents ready for delivery to users

- **Transition** - Transition the product into end user community i.e. Deploy the system in its operating environment.



The static view of RUP focuses on activities that take place during the core process. These are called workflows.

6 core workflows

- Business modeling
- Requirements
- Analysis & Design
- Implementation
- Testing
- Deployment

3 core supporting workflows

- Configuration & Change management
- Project management
- Environment

Workflow	Description
Business modelling	The business processes are modelled using business use cases.
Requirements	Actors who interact with the system are identified and use cases are developed to model the system requirements.
Analysis and design	A design model is created and documented using architectural models, component models, object models and sequence models.
Implementation	The components in the system are implemented and structured into implementation sub-systems. Automatic code generation from design models helps accelerate this process.
Test	Testing is an iterative process that is carried out in conjunction with implementation. System testing follows the completion of the implementation.
Deployment	A product release is created, distributed to users and installed in their workplace.
Configuration and change management	This supporting workflow managed changes to the system.
Project management	This supporting workflow manages the system development.
Environment	This workflow is concerned with making appropriate software tools available to the software development team.

Prepared by:
Prof. Anand Motwani
Faculty SCSE,
VIT Bhopal University

RUP Good Practices

- Six fundamental best practices are recommended:
  - Develop software iteratively
  - Manage requirements
  - Use component-based architectures
  - Visually model software
  - Verify software quality
  - Control changes to software

13. Agile Development

Definition of Agile:

An iterative and incremental (evolutionary) approach performed in a highly collaborative manner with just the right amount of ceremony to produce high quality software in a cost effective and timely manner which meets the changing needs of its stakeholders.

Core principles
"Fits just right" process
Continuous testing and validation
Consistent team collaboration
Rapid response to change
Ongoing customer involvement
Frequent delivery of working software

Agile Development Adopt disciplined, adaptive development approaches. Use continuous stakeholder feedback to deliver high quality and consumable code through use cases and a series of short, time-boxed iterations.
Business Result
Quality first
Teams build software
Minimize waste
Rapid feedback and response
One extended team
Integrated development tools
Adaptive planning

14. Personal and Team Software Process

W HAT I S P ERSONAL S OFTWARE P ROCESS (PSP)? The Personal Software Process (PSP) shows engineers how to:

- manage the quality of their projects
- make commitments they can meet
- improve estimating and planning
- reduce defects in their products PSP emphasizes the need to record and analyze the types of errors you make, so you can develop strategies eliminate them.

Prepared by:
Prof. Anand Motwani
Faculty SCSE,
VIT Bhopal University

PSP MODEL F RAMEWORK A CTIVITIES

- Planning:– isolates requirements and based on these develops both size & resource estimates. A defect estimate is made.
- High level Design:– external specification of all components. All issues are recorded and tracked.
- High level Design Review:- formal verification to uncover errors
- Development:- metrics are maintained for all important tasks & work results.
- Postmortem:- using measures & metrics collected effectiveness of process is determined an improved.

P ERSONAL S OFTWARE P ROCESS Because personnel costs constitute 70 percent of the cost of software development, the skills and work habits of engineers largely determine the results of the software development process. Based on practices found in the CMMI, the PSP can be used by engineers as a guide to a disciplined and structured approach to developing software. The PSP is a prerequisite for an organization planning to introduce the TSP.

PSP.. The PSP can be applied to many parts of the software development process, including

- small-program development
- requirement definition
- document writing
- systems tests
- systems maintenance
- enhancement of large software systems.

W HAT I S T EAM S OFTWARE P ROCESS (TSP)? The Team Software Process (TSP), along with the Personal Software Process, helps the high- performance engineer to - ensure quality software products - create secure software products - improve process management in an organization.

TSP F RAMEWORK A CTIVITIES Launch high level design Implementation Integration Test postmortem.

8 TSP.. Engineering groups use the TSP to apply integrated team concepts to the development of software-intensive systems. A launch process walks teams and their managers through
- establishing goals
- defining team roles
- assessing risks
- producing a team plan.

BENEFITS OF TSP The TSP provides a defined process framework for managing, tracking and reporting the team's progress. Using TSP, an organization can build self- directed teams that plan and track their work, establish goals, and own their processes and plans. These can be pure software teams or integrated product teams of 3 to 20 engineers. TSP will help your organization establish a mature and disciplined engineering practice that produces secure, reliable software.

Prepared by:
Prof. Anand Motwani
Faculty SCSE,
VIT Bhopal University

15. Extreme Programming

Definition
Extreme Programming (XP) is an agile software development framework that aims to produce higher quality software, and higher quality of life for the development team. XP is the most specific of the agile frameworks regarding appropriate engineering practices for software development.

When Applicable:

The general characteristics where XP is appropriate were described by Don Wells on www.extremeprogramming.org:

Dynamically changing software requirements
Risks caused by fixed time projects using new technology
Small, co-located extended development team
The technology you are using allows for automated unit and functional tests
Due to XP's specificity when it comes to it's full set of software engineering practices, there are several situations where you may not want to fully practice XP. The post When is XP Not Appropriate on the C2 Wiki is probably a good place to start to find examples where you may not want to use XP.

While you can't use the entire XP framework in many situations, that shouldn't stop you from using as many of the practices as possible given your context.

Values
The five values of XP are communication, simplicity, feedback, courage, and respect

Refer the link:

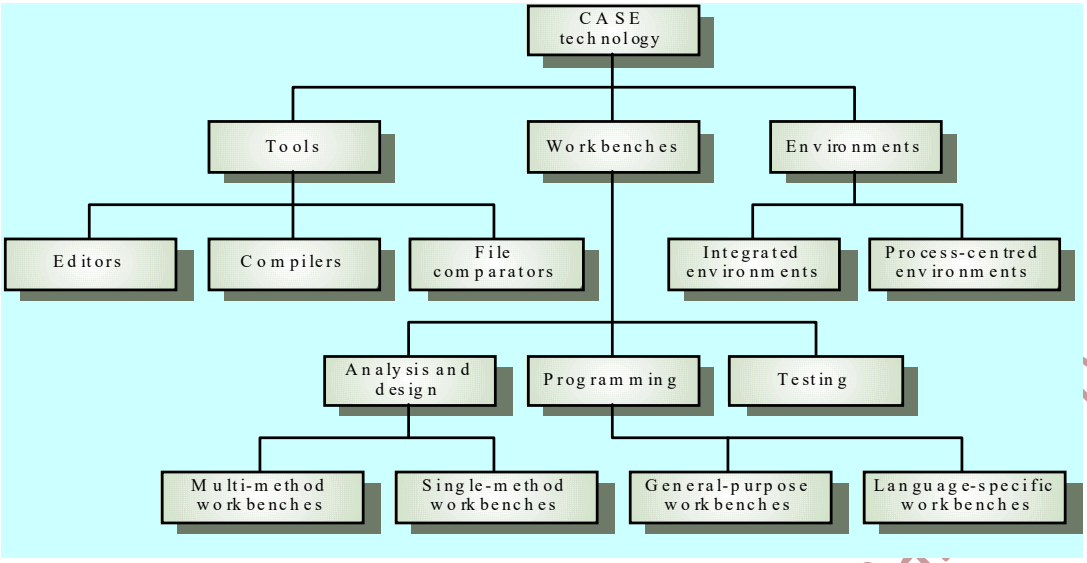
https://www.agilealliance.org/glossary/xp/#q=~(infinite~false~filters~(postType~(~'post~'aa_book~'aa_event_session~'aa_experience_report~'aa_glossary~'aa_research_paper~'aa_video)~tags~(~'xp))~searchTerm~'~sort~false~sortDirection~'asc~page~1)

15. Software Engineering Knowledge

15.1 CASE (Computer-Aided Software Engineering)

CASE tools are software systems which are designed to support routine activities in the software process such as editing design diagrams, checking diagram consistency and keeping track of program tests which have been run.

Prepared by:
Prof. Anand Motwani
Faculty SCSE,
VIT Bhopal University



15.2 Product and Process Metrics

Few Definitions

Measure -- Provides a quantitative indication of the extent, amount, dimensions, capacity, or size of some product or process attribute (1000 lines)

☐ *Metrics -- A quantitative measure of the degree to which a system, component, or process possesses a given attribute (40%)*

☐ *Software Metrics -- refers to a broad range of measurements for computer software.*

☐ *Indicator -- A metric or combination of metrics that provide insight into the software process, software project, or the product itself.*

- Two categories of software measurement
  - Direct measures of the
    - Software process (cost, effort, etc.)
    - Software product (lines of code produced, execution speed, defects reported over time, etc.)
  - Indirect measures of the
    - Software product (functionality, quality, complexity, efficiency, reliability, maintainability, etc.)
- Project metrics can be consolidated to create process metrics for an organization

Prepared by:
Prof. Anand Motwani
Faculty SCSE,
VIT Bhopal University

Size-oriented Metrics

- Derived by normalizing quality and/or productivity measures by considering the size of the software produced
- Thousand lines of code (KLOC) are often chosen as the normalization value
- Metrics include
  - Errors per KLOC - Errors per person-month
  - Defects per KLOC - KLOC per person-month
  - Dollars per KLOC - Dollars per page of documentation
  - Pages of documentation per KLOC
- Size-oriented metrics are not universally accepted as the best way to measure the software process
- Opponents argue that KLOC measurements
  - Are dependent on the programming language
  - Penalize well-designed but short programs
  - Cannot easily accommodate nonprocedural languages
  - Require a level of detail that may be difficult to achieve

Function-oriented Metrics

- Function-oriented metrics use a measure of the functionality delivered by the application as a normalization value

Most widely used metric of this type is the function point:

FP = count total * [0.65 + 0.01 * sum (value adj. factors)]

- Function point values on past projects can be used to compute, for example, the average number of lines of code per function point (e.g., 60)
- Like the KLOC measure, function point use also has proponents and opponents
- Proponents claim that
  - FP is programming language independent
  - FP is based on data that are more likely to be known in the early stages of a project, making it more attractive as an estimation approach
- Opponents claim that
  - FP requires some "sleight of hand" because the computation is based on subjective data
  - Counts of the information domain can be difficult to collect after the fact
  - FP has no direct physical meaning…it's just a number
- Relationship between LOC and FP depends upon
  - The programming language that is used to implement the software

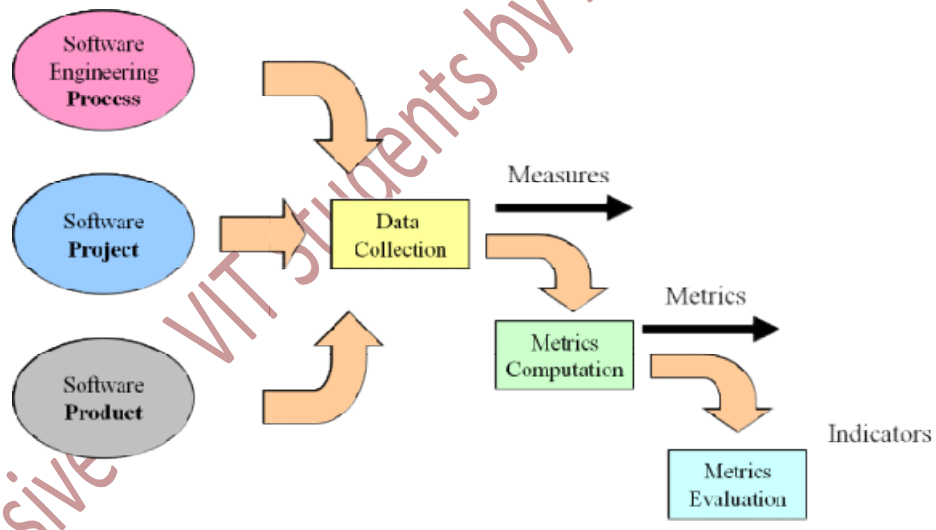
Prepared by:
Prof. Anand Motwani
Faculty SCSE,
VIT Bhopal University

- The quality of the design

- FP and LOC have been found to be relatively accurate predictors of software development effort and cost
  - However, a historical baseline of information must first be established
- LOC and FP can be used to estimate object-oriented software projects
  - However, they do not provide enough granularity for the schedule and effort adjustments required in the iterations of an evolutionary or incremental process

A rough estimate tells that average LOC to one FP in various programming languages.

Language	Average	Median	Low	High
C++	66	53	29	178
Java	55	53	9	214
Visual Basic	47	42	16	158



Prepared by:
Prof. Anand Motwani
Faculty SCSE,
VIT Bhopal University

- Given the following values, compute function point when all complexity adjustment factor (CAF) and weighting factors are average.
  User Input = 50
  User Output = 40
  User Inquiries = 35
  User Files = 6
  External Interface = 4
  Solution:
  Step-1:
  As complexity adjustment factor is average (given in question), hence,
  scale = 3.
  F = 14 * 3 = 42
  Step-2:
  CAF = 0.65 + (0.01 * 42) = 1.07
  Step-3:
  As weighting factors are also average (given in question) hence we will multiply each individual function point to corresponding values in TABLE.
  UFP = (50*4) + (40*5) + (35*4) + (6*10) + (4*7) = 628
  Step-4:
  Function Point = 628 * 1.07 = 671.96
  This is the required answer.

- **Counting Function Point (FP):**

- **Step-1:**
  F = 14 * scale
  Scale varies from 0 to 5 according to character of Complexity Adjustment Factor (CAF). Below table shows scale:
  0 - No Influence
  1 - Incidental
  2 - Moderate
  3 - Average
  4 - Significant
  5 - Essential
- **Step-2:** Calculate Complexity Adjustment Factor (CAF).
  CAF = 0.65 + (0.01 * F)
- **Step-3:** Calculate Unadjusted Function Point (UFP).

  TABLE (Required)

FUNCTION UNITS	LOW	AVG	HIGH
EI	3	4	6
EO	4	5	7

FUNCTION UNITS	LOW	AVG	HIGH
EQ	3	4	6
ILF	7	10	15
EIF	5	7	10

Multiply each individual function point to corresponding values in TABLE.

- **Step-4:** Calculate Function Point.
  FP = UFP * CAF

Measurement parameter	Count	Simple	Average	Complex	Count Total
Inputs	**50**	3	4	6	50*4 = 200
Outputs	**40**	4	5	7	40*5=200
inquiry	**35**	3	4	6	140
files	**6**	7	10	15	60
External interfaces	**4**	5	7	10	28
Total					= 628

$FP = counttotal * ¿ + 0.01 x \sum Fi ¿ ¿$

FP = counttotal x CAF

FP = 628 * [0.65 + 0.01 x (14x3)]

FP = 628 x 1.07 = 672

SOFTWARE ENGINEERING

(CSE3005)

7/18/2019

Prof. Anand Motwani
Faculty, SCSE
VIT Bhopal University

Unit II

REQUIREMENT ENGINEERING

Understanding Requirements – Establish Ground Work – Eliciting Requirements - Developing Use case- Negotiating Requirements- Validating Requirements - Requirements Modeling – Requirement Analysis –Scenario Based Modeling - UML Supplements – Data Modeling Concepts- Class Based Modeling - Flow Oriented Modeling – Creating Behavioral Modeling - Patterns For Modeling.

Aims and Objectives

- To understand of software requirements specification.
- To understand Functional and Non-Functional requirements of Software.

1. Introduction

Software Development Process: A Brief Overview

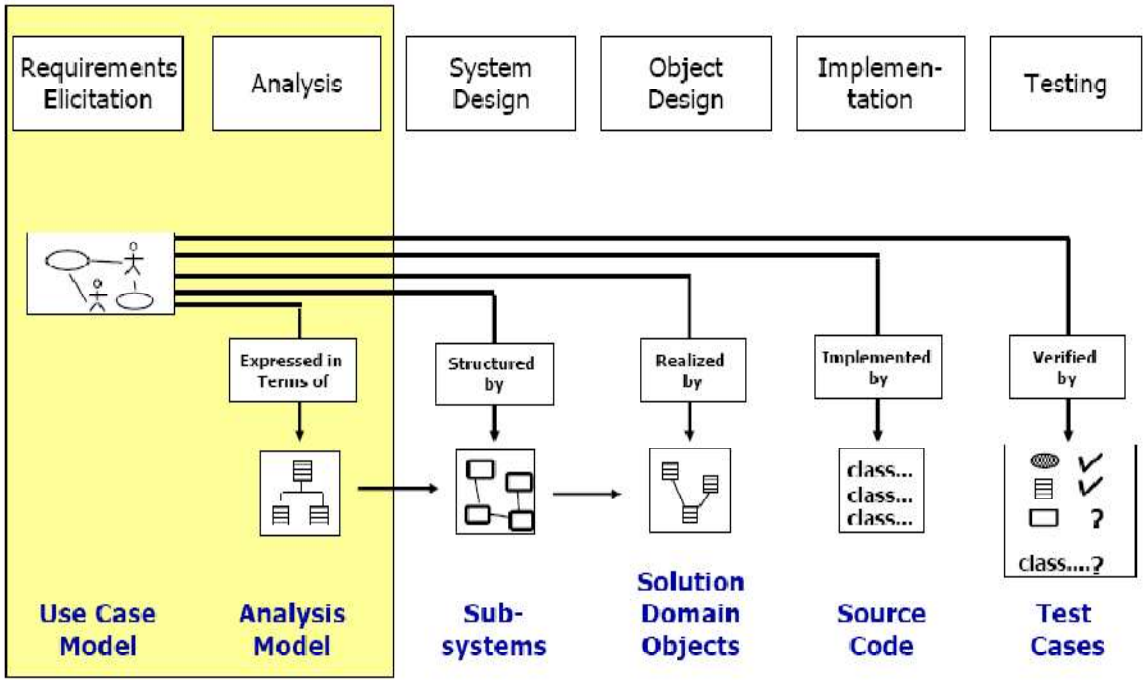


Fig. 1. Software Development Process

Prepared by:
Prof. Anand Motwani
Faculty SCSE,
VIT Bhopal University

"It Depends On Common Sense"

- *Software requirements* are documentation that completely describes the behavior that is required of the software-before the software is designed built and tested.
  - Requirements analysts (or business analysts) build software requirements specifications through *requirements elicitation*.
    - Interviews with the users, stakeholders and anyone else whose perspective needs to be taken into account during the design, development and testing of the software
    - Observation of the users at work
    - Distribution of discussion summaries to verify the data gathered in interviews

Discussion summary

- A requirements analyst can use a *discussion summary* to summarize information gathered during elicitation and validate it through a review.
- Notes gathered during the elicitation should fit into the discussion summary template
- The discussion summary outline can serve as a guide for a novice requirements analyst in leading interviews and meetings

Discussion Summary outline

- ✓ Project background
- ✓ Purpose of project
- ✓ Scope of project
- ✓ Other background information
- ✓ Perspectives
- ✓ Who will use the system?
- ✓ Who can provide input about the system?
- ✓ Project Objectives
- ✓ Known business rules
- ✓ System information and/or diagrams
- ✓ Assumptions and dependencies

Prepared by:
Prof. Anand Motwani
Faculty SCSE,
VIT Bhopal University

- ✓ Design and implementation constraints
- ✓ Risks
- ✓ Known future enhancements
- ✓ References
- ✓ Open, unresolved or 'to be determined' (TBD) issues

Requirements Analysis and Specification

Analysis is the process of specifying *what* a system will do.

- The intention is to provide a clear understanding of what the system is about and what its underlying concepts are.

The result of analysis is a *specification document*.

Requirements Engineering

- Elicitation — determining what the customer requires
- Analysis & negotiation — understanding the relationships among various customer requirements and shaping those relationships to achieve a successful result
- Requirements specification — building a tangible model of requirements
- System Modeling — building a representation of requirements that can be assessed for correctness, completeness, and consistency
- Validation — reviewing the model
- Management — identify, control and track requirements and the changes that will be made to them

Functional and Non-functional requirements

- ***Functional requirements*** define the outward behavior required of the software project.
  - The goal of the requirement is to communicate the needed behavior in as clear and unambiguous a manner as possible.
  - The behavior in the requirement can contain lists, bullets, equations, pictures, references to external documents, and any other material that will help the reader understand what needs to be implemented.
- ***Nonfunctional requirements*** define characteristics of the software which do not change its behavior.
  - Users have implicit expectations about how well the software will work.

Prepared by:
Prof. Anand Motwani
Faculty SCSE,
VIT Bhopal University

- These characteristics include how easy the software is to use, how quickly it executes, how reliable it is, and how well it behaves when unexpected conditions arise.
- The nonfunctional requirements define these aspects about the system.
    - The nonfunctional requirements are sometimes referred to as "non-behavioral requirements" or "software quality attributes"

More Explanation on Functional and Non-functional requirements

These can take a number of forms such as traditional, basic form: textual, according to some template or standard form. It includes precise statement of one function the system carries out

Example: "The system must validate the patron ID number against the patron-database before allowing the patron to check out a book."

Product requirements specify the desired characteristics that a system or subsystem must possess.

☐ Requirements for critical systems are concerned with specifying constraints on the behaviour of the executing system. Such NFRs limit the freedom of the system designers and off-the-shelf-components. Some of the components can be formulated precisely and easily quantified with Performance and Reliability. Some are difficult to quantify in terms of Usability, Adaptability.

Non-functional (product) requirements play an important role for critical systems. Failure of such systems whose causes significant economic, physical or human damage to organizations or people. So here NFRs are discussed.

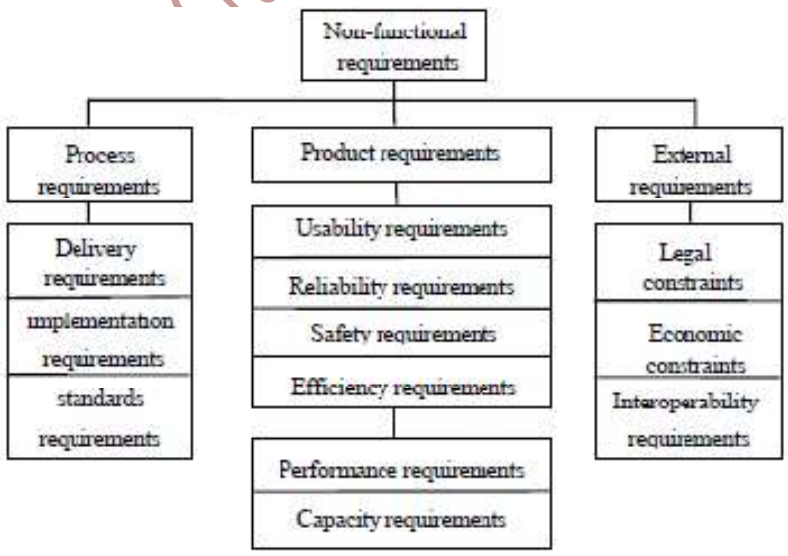


Fig. 2. Non-Functional Requirements

Prepared by:
Prof. Anand Motwani
Faculty SCSE,
VIT Bhopal University

IEEE Definition:

"non functional requirement – in software system engineering, a software requirement that describes not what the software will do, but how the software will do it, for example, software performance requirements, software external interface requirements, design constraints, and software quality attributes. Nonfunctional requirements are difficult to test; therefore, they are usually evaluated subjectively."

There is no a clear distinction between functional and nonfunctional requirements. Whether or not a requirement is expressed as a functional or a non-functional requirement may depend: on the level of detail to be included in the requirements document and the comprehension of application domain and the desired system and also on experience of developers.

Non-functional requirements define the overall qualities or attributes of the resulting system

☐ Non-functional requirements place restrictions on the product being developed, the development process, and specify external constraints that the product must meet.

☐ Examples of NFR include safety, security, usability, reliability and performance requirements.

☐ Project management issues (costs, time, schedule) are often considered as non-functional requirements as well

Some properties of a system may be expressed either as a functional or non-functional property.

Example. The system shall ensure that data is protected from unauthorized access. Conventionally a non-functional requirement (security) because it does not specify specific system functionality.

When expressed as functional requirement: The system shall include a user authorization procedure where users must identify themselves using a login name and password. Only users who are authorized in this way may access the system data. Thus Non-functional requirements may result in new functional requirements statements.

Types of Non-functional requirements

The 'IEEE-Std 830 - 1993' lists 13 non-functional requirements to be included in a Software Requirements Document. Refer Figure 2.

Classification of Non-functional requirements (Jacobson, 1999)

The FURPS+ model is proposed for the Unified Process.

☐ Functional Requirements ☐ Usability ☐ Reliability ☐ Performance☐ Supportability

☐ The FURPS+ model provides additional requirements typically also included under the general label of non-functional requirements:

Prepared by:
Prof. Anand Motwani
Faculty SCSE,
VIT Bhopal University

☐ Implementation requirements ☐ Interface requirements ☐ Operations requirements

☐ Packaging requirements ☐ Legal requirements

Requirements Elicitation

Requirements Elicitation is the process of discovering the requirements for a system by communication with customers, system users and others who have a stake in the system development. Development teams have to take the initiative.

Requirement Sources and Elicitation Techniques

Elicitation Techniques

➜ Traditional Approaches Introspection Interview/survey Group elicitation	➜ Observational approaches Protocol analysis Participant Observation (ethno methodology)
➜ Model-based approaches Goal-based: hierarchies of stakeholders' goals Scenarios: characterizations of the ways in which the system is used Use Cases: specific instances of interaction with the system	➜ Exploratory approaches Prototyping ("plan to throw one away")

Several Classifications are as Discussed in Lab and in Class.

For more details refer Power point presentation.

Use Cases

- A *use case* is a description of a specific interaction that a user may have with the system. A collection of scenarios that describe the thread of usage of a system.
- Use cases are deceptively simple tools for describing the functionality of the software.

Prepared by:
Prof. Anand Motwani
Faculty SCSE,
VIT Bhopal University

  - Use cases do not describe any internal workings of the software, nor do they explain how that software will be implemented.
  - They simply show how the steps that the user follows to use the software to do his work.
  - All of the ways that the users interact with the software can be described in this manner.

- Each scenario is described from the point-of-view of an "actor"—a person or device that interacts with the software in some way
- Each scenario answers the following questions:
  - What are the main tasks of functions performed by the actor?
  - What system information will the actor acquire, produce or change?
  - Will the actor inform the system about environmental changes?
  - What information does the actor require of the system?
  - Does the actor wish to be informed about unexpected changes

Software Requirement Analysis

requirements modelling will begin with scenario-based modelling, which results in creating a use case

- identify the "customer" and work together to negotiate "product-level" requirements
- build an analysis model
  - focus on data
  - define function
  - represent behavior
- prototype areas of uncertainty
- develop a specification that will guide design
- conduct formal technical reviews.
- **Analysis Process is depicted in Figure 3.**

Davis Principle

- Understand the problem before you begin to create the analysis model.

Prepared by:
Prof. Anand Motwani
Faculty SCSE,
VIT Bhopal University

- Develop prototypes that enable a user to understand how human-machine interaction will occur.
- Record the origin of and the reason for every requirement.
- Use multiple views of requirements.
- Prioritize requirements.
- Work to eliminate ambiguity.

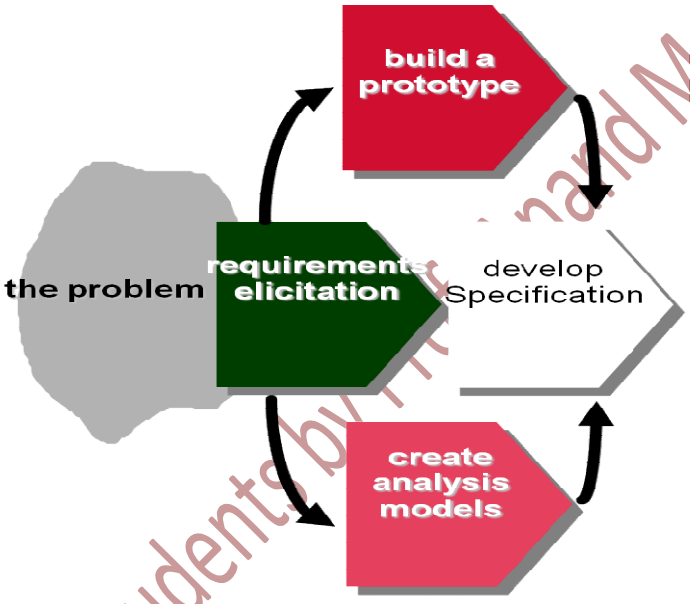
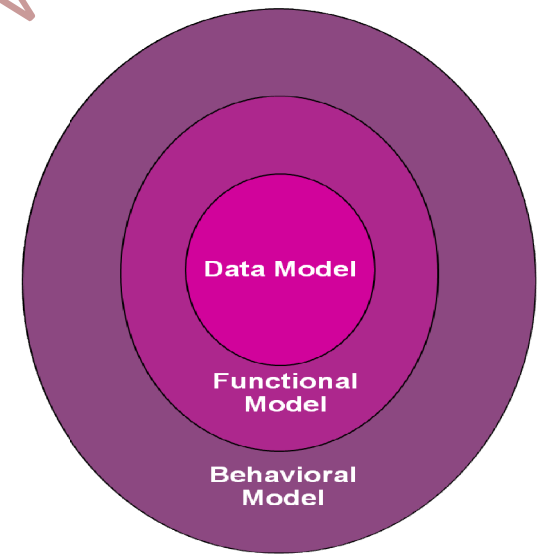


Fig. 3. Analysis Process



Prepared by:
Prof. Anand Motwani
Faculty SCSE,
VIT Bhopal University

Fig. 4. Analysis Model

Analysis Principle

I. Model the Data Domain:
- define data objects
- describe data attributes
- establish data relationships

II. Model Function
- identify functions that transform data objects
- indicate how data flow through the system
- represent producers and consumers of data

III. Model Function
- indicate different states of the system
- specify events that cause the system to change state

IV. Partition the Models
- refine each model to represent lower levels of abstraction
  - refine data objects
  - create a functional hierarchy
  - represent behavior at different levels of detail

V. Essence
- begin by focusing on the essence of the problem without regard to implementation details

Software Requirements Specification (SRS)

The *software requirements specification* (SRS) represents a complete description of the behavior of the software to be developed.

- The SRS includes:
  - A set of use cases that describe all of the interactions that the users will have with the software.
  - All of the functional requirements necessary to define the internal workings of the software: calculations, technical details, data manipulation and processing, and other specific functionality that shows how the use cases are to be satisfied
  - Nonfunctional requirements, which impose constraints on the design or implementation (such as performance requirements, quality standards or design constraints).

Prepared by:
Prof. Anand Motwani
Faculty SCSE,
VIT Bhopal University

Example of System Requirement Specification

User Requirement Definition

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

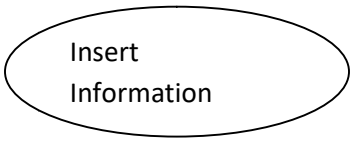
System Requirements Specification

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.

Use case Modeling

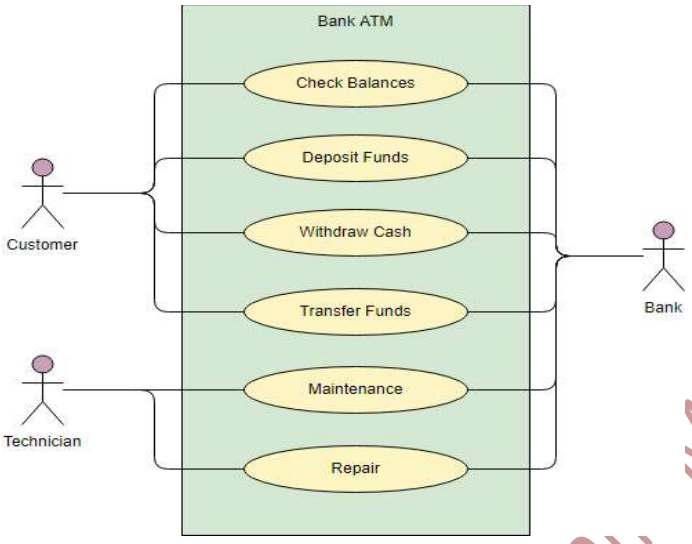
Ivar Jackobson has been credited with inventing use cases which appeared in object oriented community somewhere in 1992.

A Use Case describes a function (behaviour) provided by the system in terms of flowing events, including interaction with actors. It is initiated by an actor. Each use case has a name with entry and termination condition. Graphical Notation: An oval with the name of the use case



Use Case Model: The set of all use cases specifying the complete functionality of the system.

Prepared by:
Prof. Anand Motwani
Faculty SCSE,
VIT Bhopal University



Users, Roles and which user can invoke which use case are shown in figure.

A use case model consists of use cases and use case associations

☐ A use case association is a relationship between use cases.

The important types of use case associations: Include, Extends, Generalization

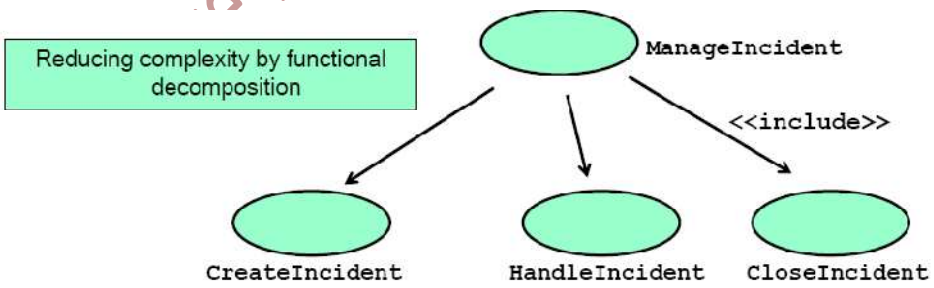
♦ Include: A use case uses another use case ("functional decomposition").

♦ Extends: A use case extends another use case.

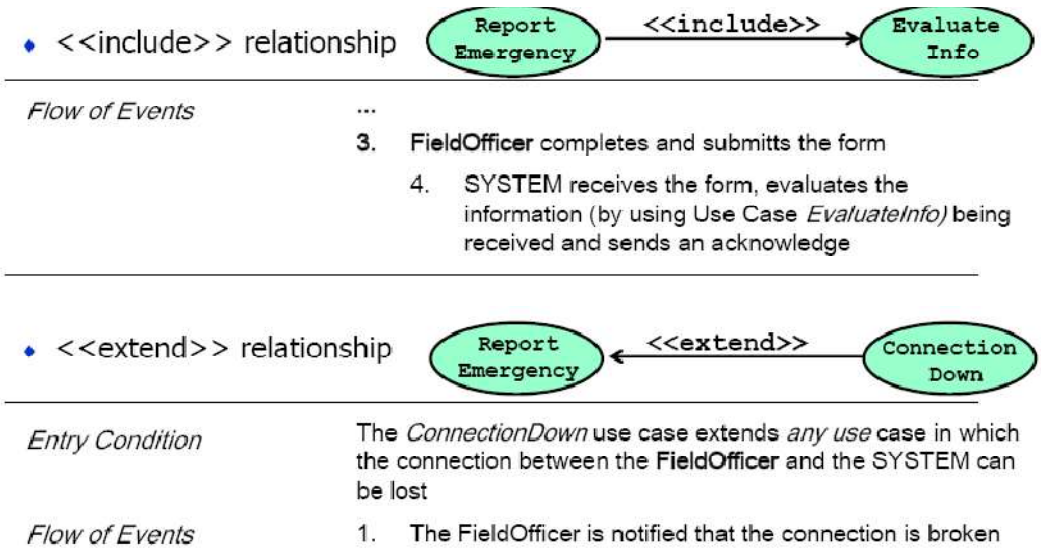
♦ Generalization: An abstract use case has different specializations.

Problem: A function in the original problem statement is too complex to be solvable immediately or the function already exists.

♦ Solution: Describe the function as the aggregation of a set of simpler functions. The associated use case is decomposed into smaller use cases.



Prepared by:
Prof. Anand Motwani
Faculty SCSE,
VIT Bhopal University



Dynamic Modeling with UML

Modeling of behavioral aspects among the participating objects is Dynamic Modeling. It shows how the behavior of a use case (or scenario) is distributed among its participating objects. In it we Tie use cases with objects. Identify new objects and classes. It helps to determine how decentralized the system is. Its purpose is to Detect and supply methods for the object model.

♦ How do we do this?

☐ Start with use case or scenario

☐ Model interaction between objects: (three types of objects i.e. Entity, boundary and control) and draw sequence diagram.

☐ Model dynamic behavior of a single object: State chart diagram.

Sequence Diagram can be read out of use cases. It is Dynamic behavior of a set of objects arranged in time sequence and good for real-time specifications and complex scenarios. It tells the Information of the life time of objects and Relationships between objects.

♦ State Chart Diagram describes the dynamic behavior of a single object. A state machine that describes the response of an object of a given class to the receipt of outside stimuli (Events).

2.5 System and Software Requirement Specifications

The goal of analysis is to discover problems, incompleteness and inconsistencies in the elicited requirements. These are then fed back to the stakeholders to resolve them through the negotiation process. Analysis is interleaved with elicitation as problems are discovered when the

Prepared by:
Prof. Anand Motwani
Faculty SCSE,
VIT Bhopal University

requirements are elicited. A problem checklist may be used to support analysis. Each requirement may be assessed against the checklist.

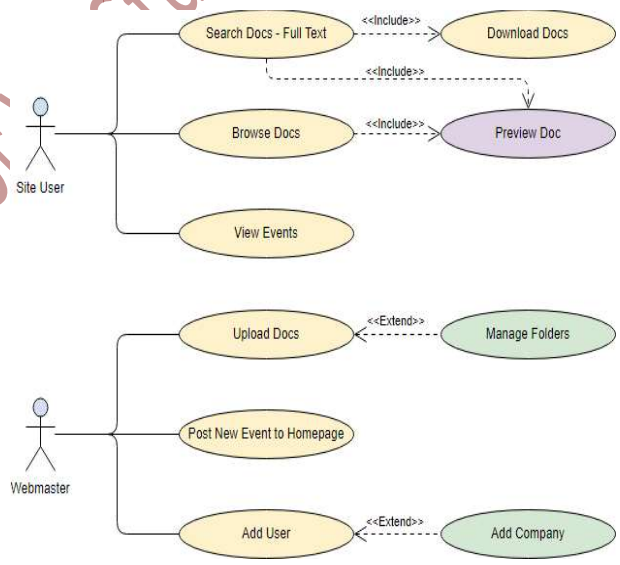
Software (System) Requirement Analysis (Summary)

1. What are the functions? → Functional Modeling
   - Create scenarios and use case diagrams
     - Talk to client, observe, get historical records, do thought experiments
2. What is the structure of the system? Create class diagrams → Object Modeling
   - Identify objects.
   - What are the associations between them? What is their multiplicity?
   - What are the attributes of the objects?
   - What operations are defined on the objects?
3. What is its behavior? → Dynamic Modeling
   - Create sequence diagrams
     - Identify senders and receivers
     - Show sequence of events exchanged between objects. Identify event dependencies and event concurrency.
   - Create state diagrams
     - Only for the dynamically interesting objects.

The above activities results in the Requirement Specification document.

Example Use Case:

The **Document Management System (DMS)** use case diagram example below shows the actors and use cases of the system. In particular, there are include and extend relationships among use cases.



Must refer to learn more about writing effective use-cases:
https://www.visual-paradigm.com/tutorials/writingeffectiveusecase.jsp

Prepared by:
Prof. Anand Motwani
Faculty SCSE,
VIT Bhopal University

Requirement Validation

Requirements validation is a critical step in the development process, usually during requirements engineering or requirements analysis and also at delivery (client acceptance test) of software.

♦ **Requirements validation criteria:**

- ✓ Complete: All possible scenarios, in which the system can be used, are described, including exceptional behavior by the user or the system.
- ✓ Consistent: There are no two functional or nonfunctional requirements that contradict each other.
- ✓ Unambiguous: Requirements can not be interpreted in mutually exclusive ways
- ✓ Correct: The requirements represent the client's view
- ✓ More Requirements validation criteria
- ✓ Realistic: Requirements can be implemented and delivered
- ✓ Verifiable: Requirements can be checked. It needs an exact description of the requirements

Problem with requirements validation: Requirements change very fast during requirements elicitation.

♦ Tool support for managing requirements:

- Store requirements in a shared repository
- Provide multi-user access
- Automatically create a system specification document from the repository
- Allow for change management
- Provide traceability throughout the project lifecycle

Analysis vs. Validation Differentiation	
Analysis (Verification) works with raw requirements as elicited from the system stakeholders	Validation works with a final draft of the requirements document i.e. with negotiated and agreed requirements
Usually incomplete and expressed in an informal and unstructured way	All known incompleteness and inconsistency are removed
Incorporation of stakeholder is important	Validation process is more concerned with the way in which the requirements are described
This tests "Have we got the right requirements"	This tests "Have we got the requirements right"

Prepared by:
Prof. Anand Motwani
Faculty SCSE,
VIT Bhopal University

Validation Inputs and Outputs

Notes on following topics will be provided into Part – II of this UNIT.

Refer PPT.

- *Data Modeling Concepts - Class Based Modeling - Flow Oriented Modeling – Creating Behavioral Modeling - Patterns For Modeling.*

Prepared by:
Prof. Anand Motwani
Faculty SCSE,
VIT Bhopal University

SOFTWARE ENGINEERING (CSE3005)

Prof. Anand Motwani
Faculty, SCSE
VIT Bhopal University

UNIT – III
(SOFTWARE DESIGN CONCEPTS)

- The Design Process 2
  - Design Concepts
  - Design Model
- Software Architecture 2
  - Architectural Styles
  - Alternatives
- Mapping To DFD 1
- Component Based Development and Design 1
- User Interface Design 1
  - Interface Analysis,
  - Interface Design.

SOFTWARE DESIGN CONCEPTS

- The systems design process allocates the requirements to either hardware or software systems by establishing an overall system architecture.
- Software design involves identifying and describing the fundamental software system abstractions and their relationships.

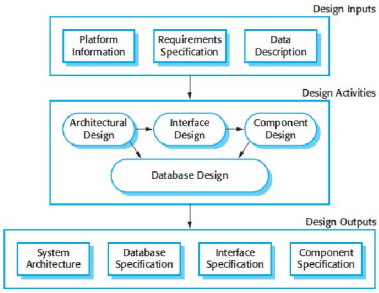
- A software design is a
  - description of the structure of the software to be implemented,
  - the data models and structures used by the system,
  - the interfaces between system components and, sometimes, the algorithms used.
  - **Note:**
  - Designers do not arrive at a finished design immediately but develop the design iteratively. They add formality and detail as they develop their design with constant backtracking to correct earlier designs.

GOAL OF DESIGN PROCESS

- The design must implement all of the explicit requirements contained in the analysis model, and it must accommodate all of the implicit requirements desired by the customer.
- The design must be a readable, understandable guide for those who generate code and for those who test and subsequently support the software.
- The design should provide a complete picture of the software, addressing the data, functional, and behavioral domains from an implementation perspective.

GENERAL MODEL OF DESIGN PROCESS

- Figure is an abstract model of this process showing the inputs to the design process, process activities, and the documents produced as outputs from this process.



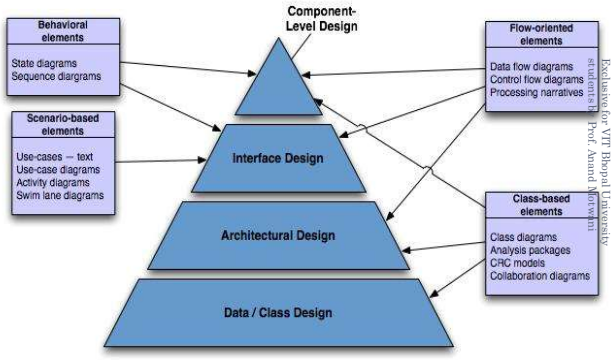
FOUR ACTIVITIES THAT MAY BE PART OF THE DESIGN PROCESS FOR INFORMATION SYSTEMS:

Design Model consists of 4 designs:

- 1. *Architectural design, where you identify the overall structure of the system, the* principal components (sometimes called sub-systems or modules), their relationships, and how they are distributed.
- 2. *Interface design, where you define the interfaces between system components.* This interface specification must be unambiguous. With a precise interface, a component can be used without other components having to know how it is implemented. Once interface specifications are agreed, the components can be designed and developed concurrently.

- 3. *Component design, where you take each system component and design how it will* operate. This may be a simple statement of the expected functionality to be implemented, with the specific design left to the programmer. Alternatively, it may be a list of changes to be made to a reusable component or a detailed design model. The design model may be used to automatically generate an implementation.
- 4. *Database design, where you design the system data structures and how these are* to be represented in a database. Again, the work here depends on whether an existing database is to be reused or a new database is to be created.

Translating Analysis → Design



Design Concepts

Design concepts

Design concepts provide the necessary framework for "to get the thing on right way".

- Abstraction
- Refinement
- Modularity
- Architecture
- Control Hierarchy
- Structural Partitioning
- Data Structure
- Software Procedure
- Information Hiding

Abstraction

- At the highest level of abstraction – a solution is stated in broad terms
- At lower level of abstraction – a more detailed description of the solution is provided.
- Two types of abstraction:
- *Procedural abstraction:* Sequence of instructions that have a specific and limited function.

Ex. Open a door

open implies long sequence of activities (e.g. walk to the door, grasp knob, turn knob and pull the door, etc).

- *Data abstraction*: collection of data that describes a data object.

Ex. Open a door. – **door** is data object.

- Data abstraction for ***door*** would encompass a set of attributes that describe the door. (E.g. door type, swing direction, opening mechanism, etc.)

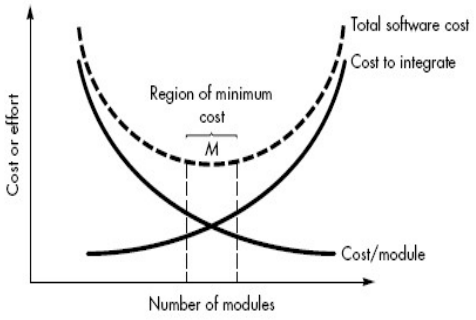
Refinement

- Refinement is actually a process of *elaboration.*
- begin with a statement of function (or description of information) 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 information.
- Refinement causes the designer to elaborate on the original statement, providing more and more detail as each successive refinement (elaboration) 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 expose low-level details as design progresses.

Modularity

- Architecture and design pattern embody modularity.
- Software is divided into separately named and addressable components, sometimes called modules, which are integrated to satisfy problem requirement.
- modularity is the single attribute of software that allows a program to be intellectually manageable
- It leads to a "divide and conquer" strategy. – it is easier to solve a complex problem when you break into a manageable pieces.
- Refer fig. that state that effort (cost) to develop an individual software module does decrease if total number of modules increase.
- However as the no. of modules grows, the effort (cost) associated with integrating the modules also grows.

Modularity and Software Cost



- Undermodularity and overmodularity should be avoided. But how do we know the vicinity of M?
- We modularize a design so that development can be more easily planned.
- Software increments can be defined and delivered.
- Changes can be more easily accommodated.
- Testing and debugging can be conducted more efficiently and long-term maintained can be conducted without serious side effects.

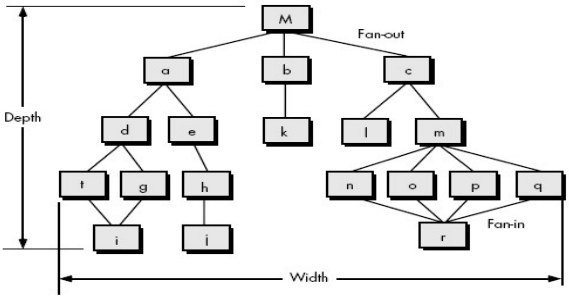
ARCHITECTURE

- Software architecture suggest " the overall structure of the software and the ways in which that structure provides conceptual integrity for a system.
- No. of different models can use to represent architecture.
  - Structural Model- represent architecture as an organized collection of components
  - Framework model – Increase level of design abstraction by identifying repeatable architectural design framework.
  - Dynamic model – address behavior of the program architecture and indicating how system or structure configuration may change as a function.
  - Process Model – focus on design of the business or technical process that the system must accommodate.
  - Functional models – used to represent the functional hierarchy of a system.

CONTROL HIERARCHY

- *Control hierarchy,* represents the organization of program components (modules) and implies a hierarchy of control.
- Different notations are used to represent control hierarchy for those architectural styles that are amenable to this representation.
- The most common is the treelike diagram that represents hierarchical control for call and return architectures.

CONTROL HIERARCHY

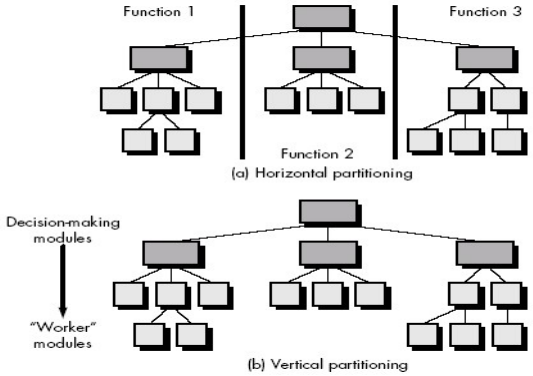


- In fig. *depth* and *width* provide an indication of the number of levels of control and overall *span of control*.
- *Fan-out* is a measure of the number of modules that are directly controlled by another module. *Fan-in* indicates how many modules directly control a given module.
- A module that controls another module is said to be *superordinate* to it, and conversely, a module controlled by another is said to be *subordinate* to the controller.

Ex. Module *M* is superordinate to modules *a, b,* and *c.*

Module *h* is subordinate to module *e*

STRUCTURAL PARTITIONING



STRUCTURE PARTITIONING

- Two types of Structure partitioning:
  - Horizontal Partitioning
  - Vertical Partitioning
- ***Horizontal partitioning*** defines separate branches of the modular hierarchy for each major program function.
- *Control modules*, represented in a darker shade are used to coordinate communication & execution between its functions.
- *Horizontal partitioning* defines three partitions—input, data transformation (often called *processing*) and output.
- Benefits of Horizontal partitioning:
  - software that is easier to test
  - software that is easier to maintain
  - propagation of fewer side effects
  - software that is easier to extend

- On the negative side (Drawback), horizontal partitioning often causes more data to be passed across module interfaces and can complicate the overall control of program flow.
- ***Vertical partitioning***, often called *factoring*, suggests that control (decision making) and work should be distributed top-down in the program structure.
- Top-level modules should perform control functions and do little actual processing work.
- Modules that reside low in the structure should be the workers, performing all input, computation, and output tasks.

- A change in a control module (high in the structure) will have a higher probability of propagating side effects to modules that are subordinate to it.
- A change to a worker module, given its low level in the structure, is less likely to cause the propagation of side effects.
- For this reason vertically partitioned structures are less likely to be susceptible to side effects when changes are made and will therefore be more maintainable.

Data Structure

- *Data structure* is a representation of the logical relationship among individual elements of data
- A *scalar item* is the simplest of all data structures. It represents a single element of information that may be addressed by an identifier.
- When scalar items are organized as a list or contiguous group, a *sequential vector* is formed.
- When the sequential vector is extended to two, three, and ultimately, an arbitrary number of dimensions, an *n-dimensional space* is created. Most common n-dimensional space is the two-dimensional matrix
- A *linked list* is a data structure that organizes contiguous scalar items, vectors, or spaces in a manner (called *nodes*) that enables them to be processed as a list.
- A *hierarchical data structure* is implemented using multilinked lists that contain scalar items, vectors, and possibly, *n*-dimensional spaces.

Software Procedure

- Software procedure focuses on the processing details of each module individually.
- Procedure must provide a precise specification of processing, including sequence of events, exact decision points, repetitive operations, and even data organization and structure.

Information Hiding

- The principle of *information hiding* suggests that modules be "characterized by design decisions that (each) hides from all others modules."
- In other words, modules should be specified and designed so that information (algorithm and data) contained within a module is inaccessible to other modules that have no need for such information.
- The intent of information hiding is to hide the details of data structure and procedural processing behind a module interface.
- It gives benefits when modifications are required during testing and maintenance because data and procedure are hiding from other parts of software, unintentional errors introduced during modification are less.

EFFECTIVE MODULAR DESIGN

- Effective modular design consist of three things:
  - Functional Independence
  - Cohesion
  - Coupling

FUNCTIONAL INDEPENDENCE

- Functional independence is achieved by developing modules with "single-minded" function and an "aversion" to excessive interaction with other modules.
- In other words - each module addresses a specific sub-function of requirements and has a simple interface when viewed from other parts of the program structure.
- Independence is important –
  - Easier to develop
  - Easier to Test and maintain
  - Error propagation is reduced
  - Reusable module.

FUNCTIONAL INDEPENDENCE

- To summarize, functional independence is a key to good design, and design is the key to software quality.
- To measure independence, have two qualitative criteria: cohesion and coupling
- *Cohesion* is a measure of the relative functional strength of a module.
- *Coupling* is a measure of the relative interdependence among modules.

COHESION

- Cohesion is a natural extension of the information hiding concept
- A cohesive module performs a single task within a software procedure, requiring little interaction with procedures being performed in other parts of a program
- Simply state, a cohesive module should (ideally) do just one thing.
- We always strive for high cohesion, although the mid-range of the spectrum is often acceptable.
- Low-end cohesiveness is much "worse" than middle range, which is nearly as "good" as high-end cohesion.
- So. designer should avoid low levels of cohesion when modules are designed.

COHESION

- When processing elements of a module are related and must be executed in a specific order, *procedural cohesion* exists.
- When all processing elements concentrate on one area of a data structure, *communicational cohesion* is present.
- High cohesion is characterized by a module that performs one distinct procedural task.

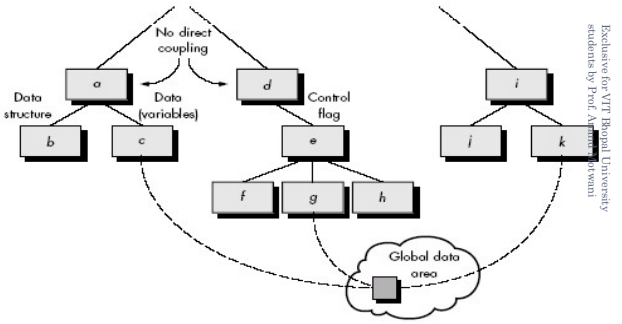
TYPES OF COHESION

- A module that performs tasks that are related logically is *logically cohesive*.
- When a module contains tasks that are related by the fact that all must be executed with the same span of time, the module exhibits *temporal cohesion*.
- At the low-end of the spectrum, a module that performs a set of tasks that relate to each other loosely, called *coincidentally cohesive*.

COUPLING

- 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 propagate through a system.
- It occur because of design decisions made when structure was developed.

COUPLING



COUPLING

- Coupling is characterized by passage of control between modules.
- "Control flag" (a variable that controls decisions in a subordinate or superordinate module) is passed between modules d and e (called control coupling).
- Relatively high levels of coupling occur when modules are communicate with external to software.
- External coupling is essential, but should be limited to a small number of modules with a structure.

- High coupling also occurs when a number of modules reference a global data area.
- Common coupling, no. of modules access a data item in a global data area
- So it does not mean "use of global data is bad". It does mean that a software designer must be take care of this thing.

EVOLUTION OF SOFTWARE DESIGN

- Early design work concentrated on criteria for the development of modular programs and methods for refining software structures in a top-down manner.
- Later work proposed methods for the translation of data flow or data structure into a design definition.
- Today, the emphasis in software design has been on software architecture and the design patterns that can be used to implement software architectures.

CHARACTERISTICS ARE COMMON TO ALL DESIGN METHODS

- A mechanism for the translation of analysis model into a design representation,
- A notation for representing functional components and their interfaces.
- Heuristics for refinement and partitioning
- Guidelines for quality assessment.

Design Quality Attributes

- Acronym FURPS –
  - Functionality
  - Usability
  - Reliability
  - Performance
  - Supportability

- Functionality – is assessed by evaluating the feature set and capabilities of the program.
  - Functions that are delivered and security of the overall system.
- Usability – assessed by considering human factors, consistency & documentation.
- Reliability – evaluated by
  - measuring the frequency and severity of failure.
  - Accuracy of output results.
  - Ability to recover from failure and predictability of the program.
- Performance - measured by processing speed, response time, resource consumption, efficiency.
- Supportability – combines the ability to extend the program (extensibility), adaptability and serviceability.

Software Architecture

"The software architecture of a program or computing system is:

- the structure or structures of the system, which comprise the software components,
- the externally visible properties of those components,
- and the relationships among them."

Software Architecture

- It is not operational software but it is representation that enables software engineer to
  - Analyze the effectiveness of design in meeting its stated requirement.
  - consider architectural alternatives at a stage when making design changes is still relatively easy,
  - reduce the risks associated with the construction of the software.

Importance of Software Architecture

- Architecture Representations is an enabler for communication between all parties (stakeholders) interested in the development
- It highlights early design decisions that will have a profound impact on all S. Engg. work that follows.
- It "constitutes a relatively small, intellectually graspable model of how the system is structured and how its components work together"

Architectural Style

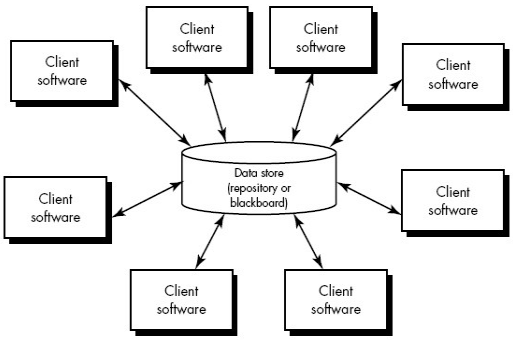
- Style describes a system category that encompasses

1. A set of *components* (e.g., a database, computational modules) that perform a function required by a system;
2. a set of *connectors* that enable "communication, co-ordinations and cooperation" 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 properties of a system

It can be represent by

- Data-centered architecture
- Data flow architecture
- Call and return architecture
- Object oriented architecture
- Layered architecture.

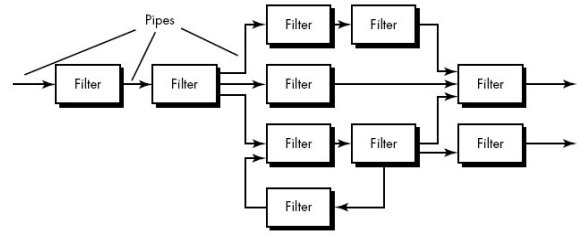
Data-centered architecture



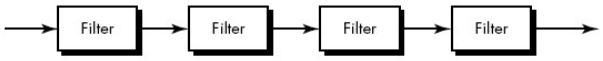
Data-centered architecture

- 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, or otherwise modify data within the store.
- Client software accesses a central repository which is in passive state (in some cases).
- client software accesses the data independent of any changes to the data or the actions of other client software.
- So, in this case transform the repository into a "Blackboard".
- A blackboard sends notification to subscribers when data of interest changes, and is thus active.
- Data-centered architectures promote *integrability*.
- Existing components can be changed and new client components can be added to the architecture without concern about other clients.
- Data can be passed among clients using the blackboard mechanism. So Client components independently execute processes

DATA FLOW ARCHITECTURE



Pipes and filters

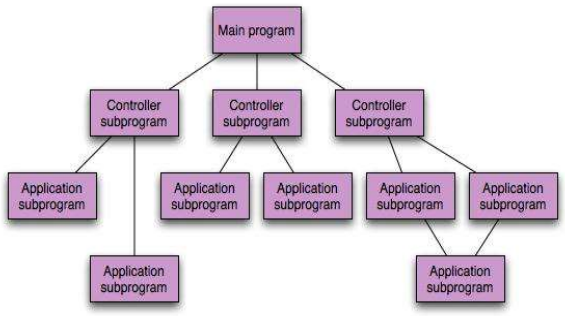


Batch Sequential

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 (Fig .)* has a set of components, called *filters,* connected by pipes that transmit data from one component to the next.
- Each filter works independently (i.e. upstream, downstream) and is designed to expect data input of a certain form, and produces data output (to the next filter) of a specified form.
- the filter does not require knowledge of the working of its neighboring filters.
- If the data flow degenerates into a single line of transforms, it is termed *batch sequential.*

CALL AND RETURN ARCHITECTURE



CALL AND RETURN ARCHITECTURE

- Architecture style enables a software designer (system architect) to achieve a program structure that is relatively easy to modify and scale.
- Two sub-styles exist within this category:

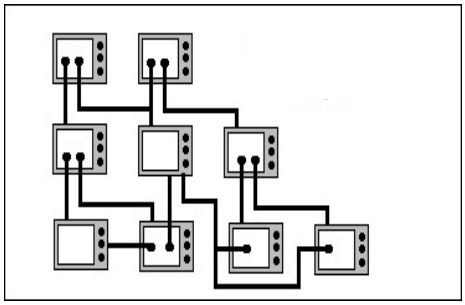
1. *Main/sub program architecture:*

- Program structure decomposes function into a control hierarchy where a "main" program invokes a number of program components, which in turn may invoke still other components.

2. *Remote procedure Call architecture:*

- The components of a main program/subprogram architecture are distributed across multiple computers on a network

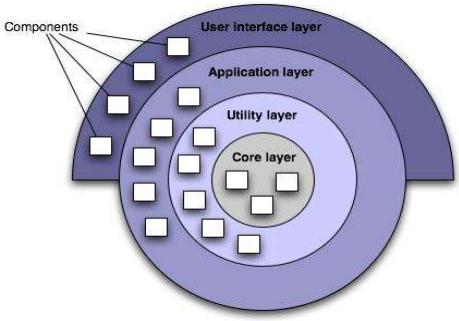
OBJECT-ORIENTED ARCHITECTURE



OBJECT-ORIENTED ARCHITECTURE

- The object-oriented paradigm, like the abstract data type paradigm from which it evolved, emphasizes the bundling of data and methods to manipulate and access that data (Public Interface).
- Components of a system summarize data and the operations that must be applied to manipulate the data.
- Communication and coordination between components is accomplished via message passing.

LAYERED ARCHITECTURE

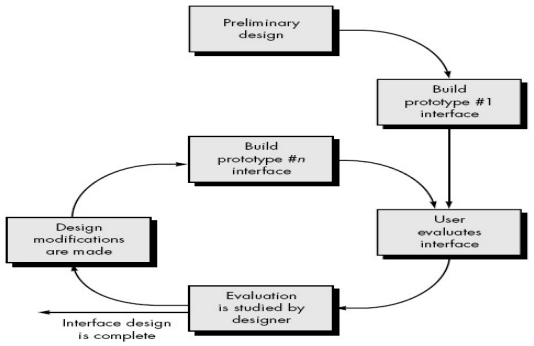


- A number of different layers are defined, each accomplishing operations that progressively become closer to the machine instruction set.
- At the outer layer, components examine user interface operations.
- At the inner layer, components examine operating system interfacing.
- Intermediate layers provide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

User Interface Design

- User interface design creates an effective communication medium between a human and a computer.
- Following a set of interface design principles, design identifies interface objects and actions and then creates a screen layout that forms the basis for a user interface prototype.

Design Evaluation



user interface evaluation cycle

- After the design model has been completed, a first-level prototype is created.
- The prototype is evaluated by the user, who provides the designer with direct comments about the efficiency of the interface.
- In addition, if formal evaluation techniques are used (e.g., questionnaires, rating sheets), the designer may extract information from these data.
- Design modifications are made based on user input and the next level prototype is created.
- The evaluation cycle continues until no further modifications to the interface design are necessary.
- The prototyping approach is effective, but is it possible to evaluate the quality of a user interface before a prototype is built?
- If potential problems uncovered and corrected early, the number of loops through the evaluation cycle will be reduced and development time will shorten.

- Evaluation criteria can be applied during early design reviews:

1. The length and complexity of the written specification of the system and its interface provide an indication of the amount of learning required by users of the system.
2. The number of user tasks specified and the average number of actions per task provide an indication of interaction time and the overall efficiency of the system.
3. The number of actions, tasks, and system states indicated by the design model imply the memory load on users of the system.
4. Interface style, help facilities, and error handling protocol provide a general indication of the complexity of the interface and the degree to which it will be accepted by the user.

- Once the first prototype is built, the designer can collect a variety of qualitative and quantitative data that will assist in evaluating the interface.
- To collect qualitative data, questionnaires can be distributed to users of the prototype.
- Questions can be all
  - simple yes/no response,
  - numeric response,
  - scaled (subjective) response,
  - percentage (subjective) response.
  - Likert scale (e.g. strongly disagree, somewhat agree).
  - Open-minded.

EXAMPLE

1. Were the icons self-explanatory? If not, which icons were unclear?
2. Were the actions easy to remember and to invoke?
3. How many different actions did you use?
4. How easy was it to learn basic system operations (scale 1 to 5)?
5. Compared to other interfaces you've used, how would this rate—top 1%, top 10%, top 25%, top 50%, bottom 50%?

QUIZ

1. What is outside in design?

- a) Interaction design works mainly from the computational goals mandated in requirements specifications , to the realization as interactions between cooperating program units
- b) Many program interactions do not give rise to interesting or complicated program component interactions
- c) All of the mentioned
- d) None of the mentioned

2. What is controller ?

- a) It is a program component that makes decisions and directs other components
- b) It is a way that decision making is distributed among program components
- c) All of the mentioned
- d) None of the mentioned

3. Match the following

GROUP I	GROUP II
(P) Service oriented computing	(1) Interoperability
(Q) Heterogeneous communicating systems	(2) BPMN
(R) Information representation	(3) Publish-find-bind
(S) Process description	(4) XML

- **(A)** P-1, Q-2, R-3, S-4
  (B) P-3, Q-4, R-2, S-1
  (C) P-3, Q-1, R-4, S-2
  (D) P-4, Q-3, R-2, S-1

Mapping To DFD

- Already discussed and performed practical in class.
- For sample example refer the following link:
- https://study.com/academy/lesson/software-architecture-design-transform-mapping.html

Component Based Development and Design

- Component-based development (CBD) is a procedure that accentuates the design and development of computer-based systems with the help of reusable software components.
- techniques involve:
  - procedures for developing software systems by choosing ideal off-the-shelf components
  - then assembling them using a well-defined software architecture.

Component Based Development and Design

- Key Goals
- The CBD intends to deliver better quality and output. The key goals are as follows:
- Save time and money when building large and complex systems: Developing complex software systems with the help of off-the-shelf components helps reduce software development time substantially. Function points or similar techniques can be used to verify the affordability of the existing method.
- Enhance the software quality
- Detect defects within the systems: The CBD strategy supports fault detection by testing the components; however, finding the source of defects is challenging in CBD.

CBD

CBD: specific routines:

- Component development
- Component publishing
- Component lookup as well as retrieval
- Component analysis
- Component assembly

CBD

- There are many standard component frameworks such as COM/DCOM, JavaBean, EJB, CORBA, .NET, web services, and grid services.
- These technologies are widely used in local desktop GUI application design such as graphic JavaBean components, MS ActiveX components, and COM components which can be reused by simply drag and drop operation.

USER INTERFACE DESIGN

Interface design

- It defines a set of **interface** objects, actions, and their screen representations that enable a **user** to perform all defined tasks in a manner that meets every usability objective defined for the system.
- Interface design is concerned with specifying the detail of the interface to an object or to a group of objects.

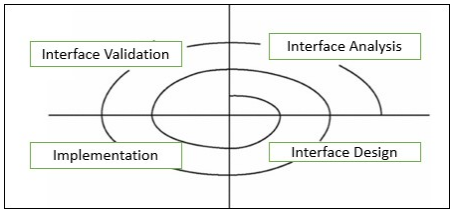
FOUR LEVELS IN UI DESIGN

- **The conceptual level** – It describes the basic entities considering the user's view of the system and the actions possible upon them.
- **The semantic level** – It describes the functions performed by the system i.e. description of the functional requirements of the system, but does not address how the user will invoke the functions.
- **The syntactic level** – It describes the sequences of inputs and outputs required to invoke the functions described.
- **The lexical level** – It determines how the inputs and outputs are actually formed from primitive hardware operations.

Stages in development of UI

User Interface Development Process

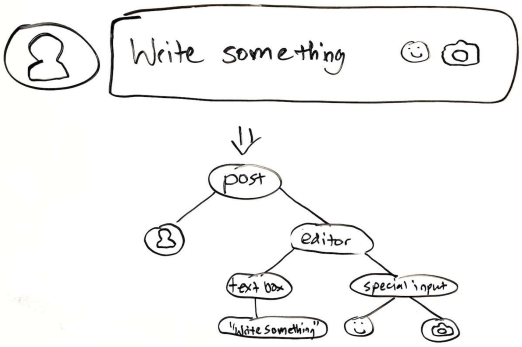
- It follows a spiral process as shown in the following diagram –
- It is iterative process.



Design & Analysis User Interface

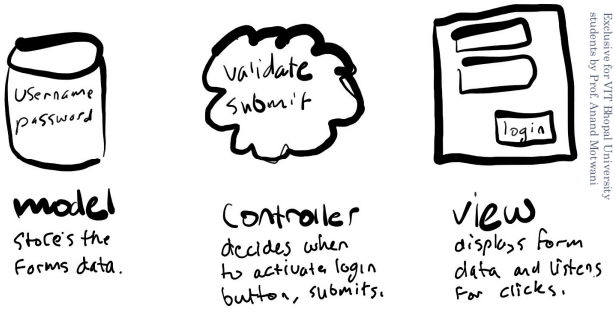
- It starts with task analysis which understands the user's primary tasks and problem domain. It should be designed in terms of User's terminology and outset of user's job rather than programmer's.
- To perform user interface analysis, the practitioner needs to study and understand four elements –
  - The **users** who will interact with the system through the interface
  - The **tasks** that end users must perform to do their work
  - The **content** that is presented as part of the interface
  - The **work environment** in which these tasks will be conducted

Example of FB UI



Model View Controller (MVC)

- A nearly ubiquitous way to implement the application is to follow a model-view-controller architecture.
- This architecture divides a user interface into three parts:



- The **MODEL** stores the *data* that a user interface is presenting. For example, in the figure above, the model stores the username and password a user is currently entering. (In many user interface toolkits, this wouldn't be stored in a separate database, but in the text field object in memory).
- The **VIEW** visualizes the data in the model. For example, in the figure above, the view is the form itself and the controls on it, including the text fields and login button. The view's job is to render the controls, listen for input from the user (e.g., clicking the login button), and display any output the controller decides to provide (e.g., form validation feedback).
- The **CONTROLLER** mediates implements the logic of the user interface. In our login form example above, that includes validating the username and password (e.g., neither can be empty), and submitting the information when the login button is pressed. The controller gets and stores data when necessary, and tells the view to update itself as the model changes.

77

SOFTWARE ENGINEERING

(CSE3005)

9/21/2019

Prof. Anand Motwani

Faculty, SCSE

VIT Bhopal University

Unit IV

IMPLEMENTATION AND TESTING

Software Implementation Techniques: Coding Practices – Refactoring - Software Testing Fundamentals - Types of Testing: Unit, Integration and System testing - Testing Strategies: Black box and White box testing - System testing and debugging.

1. Software Implementation Techniques (Introduction):

- **Best coding practices** are a set of informal rules that the software development community has learned over time which can help improve the quality of software.
- Many computer programs remain in use for far longer than the original authors ever envisaged (sometimes 40 years or more) so any rules need to facilitate both initial development and subsequent maintenance and enhancement by people other than the original authors.
- Ninety-ninety rule, Tim Cargill is credited with this explanation as to why programming projects often run late.
- The size of a project or program has a significant effect on error rates, programmer productivity, and the amount of management needed
    - Maintainability.
    - Dependability.
    - Efficiency.
    - Usability.

Prepared by:
Prof. Anand Motwani
Faculty SCSE,
VIT Bhopal University

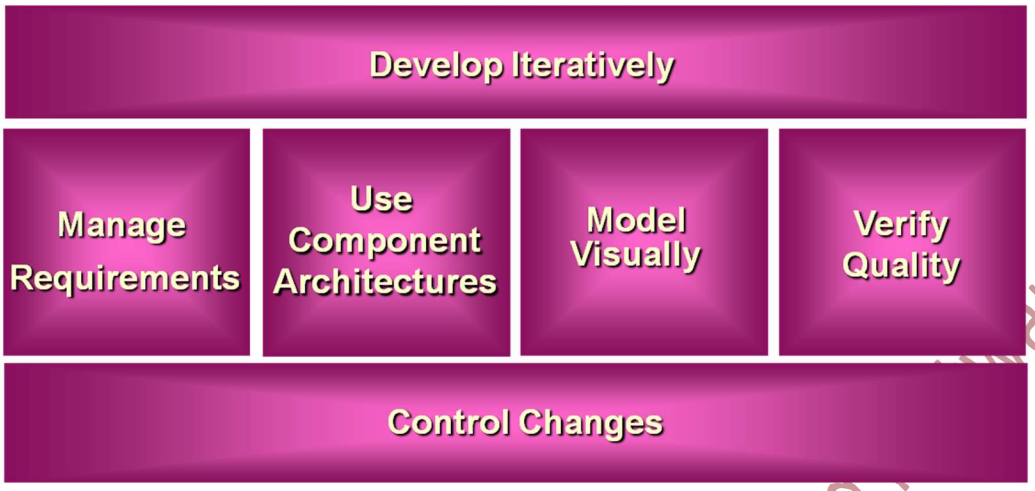


Fig. 1. Best Practices of Software Engineering

1.1 Coding Practices

Refactoring

- Refactoring is usually motivated by noticing a code smell. For example the method at hand may be very long, or it may be a near duplicate of another nearby method. Once recognized, such problems can be addressed by *refactoring* the source code, or transforming it into a new form that behaves the same as before but that no longer "smells".
- For a long routine, one or more smaller subroutines can be extracted; or for duplicate routines, the duplication can be removed and replaced with one shared function. Failure to perform refactoring can result in accumulating technical debt.
- There are two general categories of benefits to the activity of refactoring.
    - Maintainability
    - Extensibility
- Before applying a refactoring to a section of code, a solid set of automatic unit tests is needed. The tests are used to demonstrate that the behavior of the module is correct before the refactoring.
- If it inadvertently turns out that a test fails, then it's generally best to fix the test first, because otherwise it is hard to distinguish between failures introduced by refactoring and failures that were already there. After the refactoring, the tests are run again to verify the refactoring didn't break the tests.

Prepared by:
Prof. Anand Motwani
Faculty SCSE,
VIT Bhopal University

- Of course, the tests can never prove that there are no bugs, but the important point is that this process can be cost-effective: good unit tests can catch enough errors to make them worthwhile and to make refactoring safe enough.

1.2 Software Testing Fundamentals

- Testing is intended to show that a program does what it is intended to do and to discover program defects before it is put into use.
- When you test software, you execute a program using artificial data. You check the results of the test run for errors, anomalies, or information about the program's non-functional attributes.

The testing process has two distinct goals:

1. To demonstrate to the developer and the customer that the software meets its requirements. For custom software, this means that there should be at least one test for every requirement in the requirements document. For generic software products, it means that there should be tests for all of the system features, plus combinations of these features, that will be incorporated in the product release.

- This first goal leads to validation testing, where you expect the system to perform correctly using a given set of test cases that reflect the system's expected use.

2. To discover situations in which the behavior of the software is incorrect, undesirable, or does not conform to its specification. These are a consequence of software defects. Defect testing is concerned with rooting out undesirable system behavior such as system crashes, unwanted interactions with other systems, incorrect computations, and data corruption.

- The second goal leads to defect testing, where the test cases are designed to expose defects. The test cases in defect testing can be deliberately obscure and need not reflect how the system is normally used.

Prepared by:
Prof. Anand Motwani
Faculty SCSE,
VIT Bhopal University

Of course, there is no definite boundary between these two approaches to testing. During validation testing, you will find defects in the system; during defect testing, some of the tests will show that the program meets its requirements.

Edsger Dijkstra, an early contributor to the development of software engineering, eloquently stated (Dijkstra et al., 1972):

"Testing can only show the presence of errors, not their absence."

- Testing is part of a broader process of software verification and validation (V & V).
- Verification and validation are not the same thing, although they are often confused.
- Barry Boehm, a pioneer of software engineering, succinctly expressed the difference between them (Boehm, 1979):
- ■ 'Validation: Are we building the right product?'
- ■ 'Verification: Are we building the product right?'

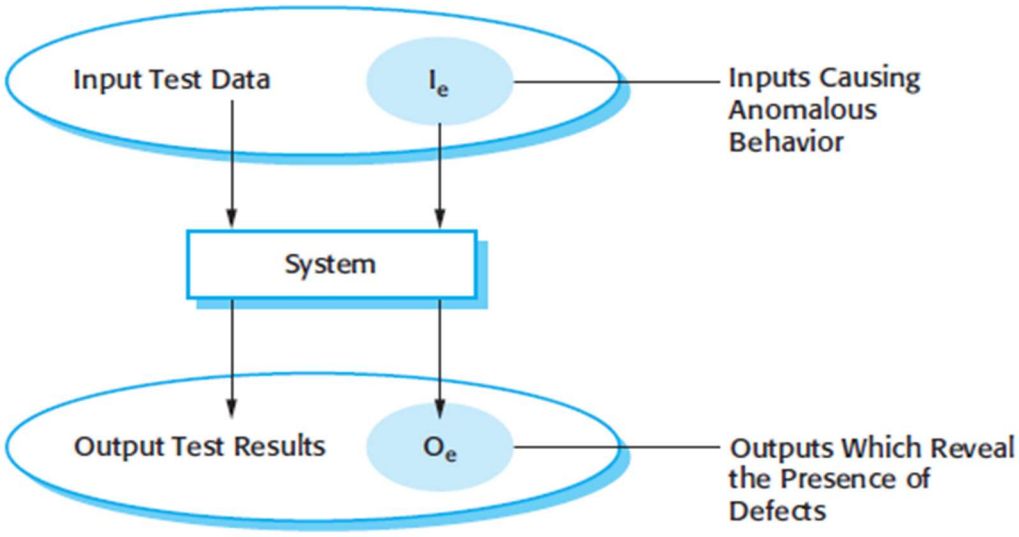


Fig. 2.

Prepared by:
Prof. Anand Motwani
Faculty SCSE,
VIT Bhopal University

Test planning is concerned with scheduling and resourcing all of the activities in the testing process. It involves:

- Defining the testing process, taking into account the people and the time available. Usually, a test plan will be created, which defines what is to be tested, the predicted testing schedule, and how tests will be recorded.
- For critical systems, the test plan may also include details of the tests to be run on the software.

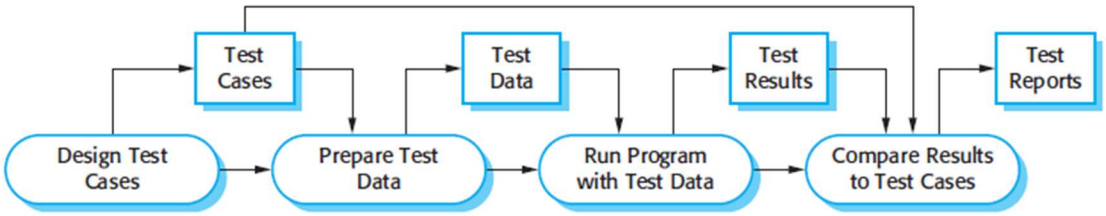


Fig. 3. Test plan

1.3 More testing fundamentals (*System testing and debugging)*

Normally, programmers carry out some testing of the code they have developed. This often reveals program defects that must be removed from the program. This is called **debugging**.
Defect testing and debugging are different processes. *Testing establishes the existence of defects. Debugging is concerned with locating and correcting these defects.*

Debugging process: When you are debugging, you have to generate hypotheses about the observable behavior of the program then test these hypotheses in the hope of finding the fault that caused the output anomaly. Testing the hypotheses may involve tracing the program code manually. It may require new test cases to localize the problem. Interactive debugging tools, which show the intermediate values of program variables and a trace of the statements executed, may be used to support the debugging process.

Prepared by:
Prof. Anand Motwani
Faculty SCSE,
VIT Bhopal University

1.4 Stages and Types of Testing:

Except for small programs, systems should not be tested as a single, monolithic unit. Figure below shows a three-stage testing process in which system components are tested then the integrated system is tested and, finally, the system is tested with the customer's data. Ideally, component defects are discovered early in the process, and interface problems are found when the system is integrated. However, as defects are discovered, the program must be debugged and this may require other stages in the testing process to be repeated. Errors in program components, say, may come to light during system testing. The process is therefore an iterative one with information being fed back from later stages to earlier parts of the process.

The stages in the testing process are:

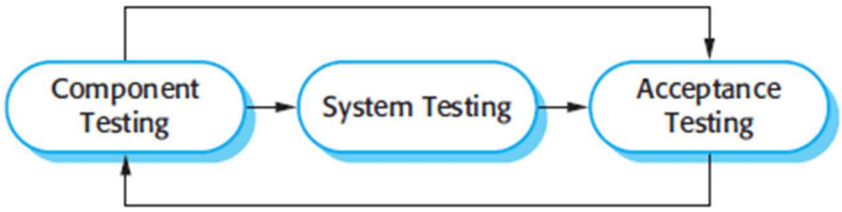


Fig. 4. Stages of Testing

1. Development (Unit & Component) testing: The components making up the system are tested by the people developing the system. Each component is tested independently, without other system components. Components may be simple entities such as functions or object classes, or may be coherent groupings of these entities. Test automation tools, such as JUnit (Massol and Husted, 2003), that can re-run component tests when new versions of the component are created, are commonly used.

During development, testing may be carried out at three levels of granularity:

1. Unit testing, where individual program units or object classes are tested. Unit testing should focus on testing the functionality of objects or methods. Unit testing is the

Prepared by:
Prof. Anand Motwani
Faculty SCSE,
VIT Bhopal University

process of testing program components, such as methods or object classes. Individual functions or methods are the simplest type of component. Your tests should be calls to these routines with different input parameters. Example of Unit testing is given at the end of the document.

2. Component testing, where several individual units are integrated to create composite components. Component testing should focus on testing component interfaces.

3. System Testing: Refer below

2. System testing: System components are integrated to create a complete system. The individual program units or programs are integrated and tested as a complete system to ensure that the software requirements have been met. This process is concerned with finding errors that result from unanticipated interactions between components and component interface problems. It is also concerned with showing that the system meets its functional and non-functional requirements, and testing the emergent system properties. For large systems, this may be a multi-stage process where components are integrated to form subsystems that are individually tested before these sub-systems are themselves integrated to form the final system.

- *This testing is conducted on a complete, integrated system to evaluate the system's compliance or reliability with its specified requirements.*

3. Acceptance testing: This is the final stage in the testing process before the system is accepted for operational use. The system is tested with data supplied by the system customer rather than with simulated test data. Acceptance testing may reveal errors and omissions in the system requirements definition, because the real data exercise the system in different ways from the test data. Acceptance testing may also reveal requirements problems where the system's facilities do not really meet the user's needs or the system performance is unacceptable.
Another definition:

- *It is a Formal testing with respect to user needs, requirements, and business processes conducted to determine whether a system satisfies the acceptance criteria.*

Acceptance testing is sometimes called 'alpha testing'. Custom systems are developed for a single client. The alpha testing process continues until the system developer and the client

Prepared by:
Prof. Anand Motwani
Faculty SCSE,
VIT Bhopal University

agree that the delivered system is an acceptable implementation of the requirements. ***Alpha*** *Testing is conducted by a team of highly skilled testers at development site*

When a system is to be marketed as a software product, a testing process called 'beta testing' is often used.

Beta testing involves delivering a system to a number of potential customers who agree to use that system. They report problems to the system developers. This exposes the product to real use and detects errors that may not have been anticipated by the system builders. After this feedback, the system is modified and released either for further beta testing or for general sale.

- *Beta Testing is always conducted in Real Time environment by customers or end users at their own site.*

Prepared by:
Prof. Anand Motwani
Faculty SCSE,
VIT Bhopal University

1.5 SDLC – V model

- The V-model is a type of SDLC model where process executes in a sequential manner in V-shape.
- It is also known as Verification and Validation model. It is based on the association of a testing phase for each corresponding development stage.
- Development of each step directly associated with the testing phase. The next phase starts only after completion of the previous phase i.e. for each development activity, there is a testing activity corresponding to it.

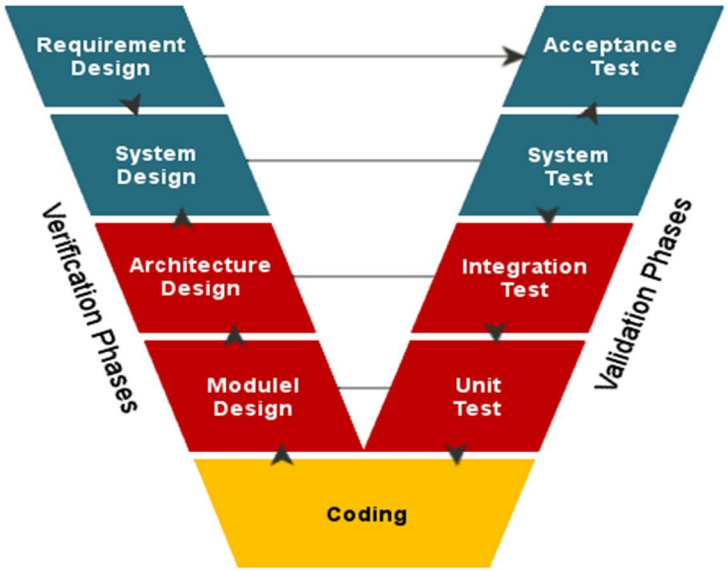


Fig. 5. SDLC-V model

- *In this execution of processes happens in a sequential manner in V-shape. It is also known as Verification and Validation Model. V-Model is an extension of the Waterfall Model and is based on association of a testing phase for each corresponding development stage.*

Prepared by:
Prof. Anand Motwani
Faculty SCSE,
VIT Bhopal University

Figure below is another illustration of how test plans are the link between testing and development activities. This is sometimes also called the V-model of development (turn it on its side to see the V).

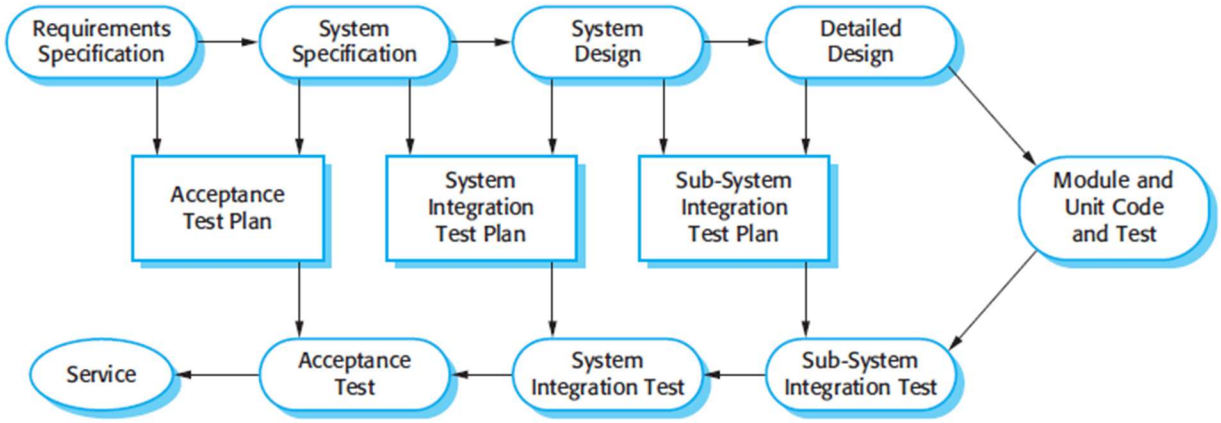


Fig. 6. Testing phases in a plan-driven software process

Other Important Testing types and strategies:

- **Regression Testing:** Retesting of a software system to confirm that changes made to few parts of the codes has not any side effects on existing system functionalities. The emphasis on regression testing in agile methods lowers the risk of introducing new errors through refactoring. Regression testing involves running test sets that have successfully executed after changes have been made to a system. The regression test checks that these changes have not introduced new bugs into the system and that the new code interacts as expected with the existing code. Regression testing is very expensive and often impractical when a system is manually tested, as the costs in time and effort are very high. In such situations, you have to try and choose the most relevant tests to re-run and it is easy to miss important tests. Automated tests and a testing framework, such as JUnit, radically simplify regression testing as the entire test set can be run automatically each time a change is made.

Note: All the above testing types come under the level of Functional Testing. Another level of testing is given below:

Prepared by:
Prof. Anand Motwani
Faculty SCSE,
VIT Bhopal University

- **Non-Functional Testing:** *Non-Functional testing is designed to figure out if your product will provide a good user experience.*
    - ***Performance Testing:*** *To evaluate the performance of components of a particular system under a particular workload.*
    - ***Load Testing:*** *Testing the behaviour of the system under a specific load or to get the breakeven point where system starts downgrading it's performance.*
    - ***Stress Testing:*** *It is performed to find the upper limit capacity of the system and also to determine how the system performs if the current load goes well above the expected maximum.*
    - ***Usability Testing:*** *Testing to determine the extent to which the software product is understood, easy to learn, easy to operate and attractive to the users under specified conditions.*
    - ***Security Testing:*** *This intends to uncover vulnerabilities of the system and determine that its data and resources are protected from possible intruders.*
    - ***Portability Testing:*** *Software reliability is the probability that software will work properly in a specified environment and for a given amount of time.*

1.6 Testing Strategies

- **Black-box testing:** An approach to testing where the testers have no access to the source code of a system or its components. The tests are derived from the system specification. Release testing is usually a black-box testing process where tests are derived from the system specification.

- **White-box or structural testing**:

  It is an approach to program testing where the tests are based on knowledge of the structure of the program and its components. Access to source code is essential for white-box testing. It is a systematic approach to testing where knowledge of the program source code is used to design defect tests. The aim is to design tests that provide some

Prepared by:
Prof. Anand Motwani
Faculty SCSE,
VIT Bhopal University

level of program coverage. That is, the set of tests should ensure that every logical path through the program is executed, with the consequence that each program statement is executed at least once.

1.7 Ways to perform Testing

1. Manual Testing: Test Cases executed manually. In this method the tester plays an important role of end user and verifies that all the features of the application are working correctly. The tester manually executes test cases without using any automation tools. The tester prepares a test plan document which describes the detailed and systematic approach to testing of software applications. Test cases are planned to cover almost 100% of the software application. As manual testing involves complete test cases it is a time consuming test.

2. Automation (Automated) Testing: Testing performed with the help of automation tools. (Most popular testing automation tool is Selenium). Agile methods recommend that very frequent system builds should be carried out with automated testing to discover software problems.

1.8 Test Artifacts

- Test Basis: It is the information needed in order to start the test analysis and create our Test Cases.
- Test Case Specification: A document described detailed summary of what scenarios will be tested, how they will be tested, how often they will be tested.
- Test Scenario: It is also called Test Condition or Test Possibility means any functionality that can be tested.
- Test Case: A set of inputs, execution preconditions, and expected outcomes developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement.
- Test Script - Is a collection of test cases for the software under test (manual or automated)

Prepared by:
Prof. Anand Motwani
Faculty SCSE,
VIT Bhopal University

Test case Example:

- To test a method called 'catWhiteSpace' in a 'Paragraph' object that, within the paragraph, replaces sequences of blank characters with a single blank character. Identify testing partitions for this example and derive a set of tests for the 'catWhiteSpace' method.

Solution:

Testing partitions / scenarios are:

- Strings with only single blank characters
- Strings with sequences of blank characters in the middle of the string
- Strings with sequences of blank characters at the beginning/end of string

Examples of tests:

- The quick brown fox jumped over the lazy dog (only single blanks)
- The quick brown fox jumped over the lazy dog (different numbers of blanks in the sequence)
- The quick brown fox jumped over the lazy dog (1st blank is a sequence)
- The quick brown fox jumped over the lazy dog (Last blank is a sequence)
- The quick brown fox jumped over the lazy dog (2 blanks at beginning)
- The quick brown fox jumped over the lazy dog (several blanks at beginning)
- The quick brown fox jumped over the lazy dog (2 blanks at end)
- The quick brown fox jumped over the lazy dog (several blanks at end)
- Etc.

Prepared by:
Prof. Anand Motwani
Faculty SCSE,
VIT Bhopal University

Example Test Script – 1:

- System Test of input of numeric month into data field

Ref.	Field/Button	Action	Input	Expected Result	Pass/Fail
001	Month	Enter Data	0	Data rejected. Error Message 'Invalid Month'	Fail
002	Month	Enter Data	1	Data Accepted, January Displayed	Pass
003	Month	Enter Data	06	Data Accepted, June Displayed	Pass
004	Month	Enter Data	12	Data Accepted, December Displayed	Pass
005	Month	Enter Data	13	Data rejected. Error Message 'Invalid Month'	Fail

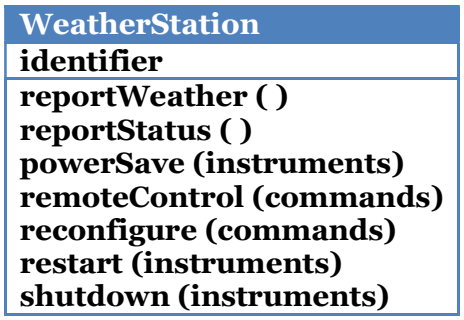
Unit testing Example:

When you are testing object classes, you should design your tests to provide coverage of all of the features of the object. This means that you should:

• test all operations associated with the object;

• set and check the value of all attributes associated with the object;

• put the object into all possible states. This means that you should simulate all events that cause a state change.



Prepared by:
Prof. Anand Motwani
Faculty SCSE,
VIT Bhopal University