

Introduction

What is Software Engineering?

- Engineering approach to develop software.
 - Building Construction Analogy.
 - Systematic collection of past experience:
 - Techniques,
 - Methodologies,
 - Guidelines.



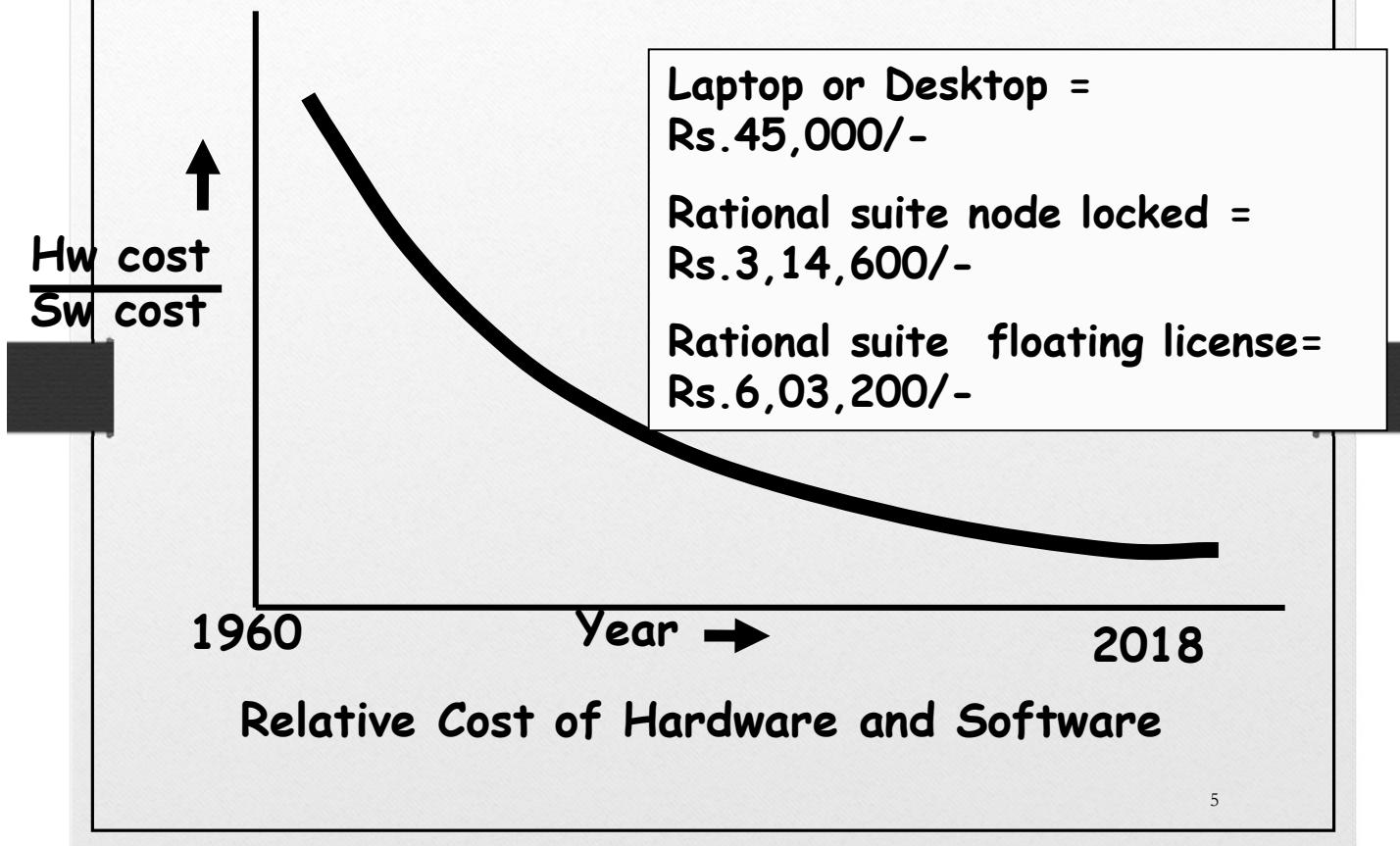
IEEE Definition

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

Software Crisis

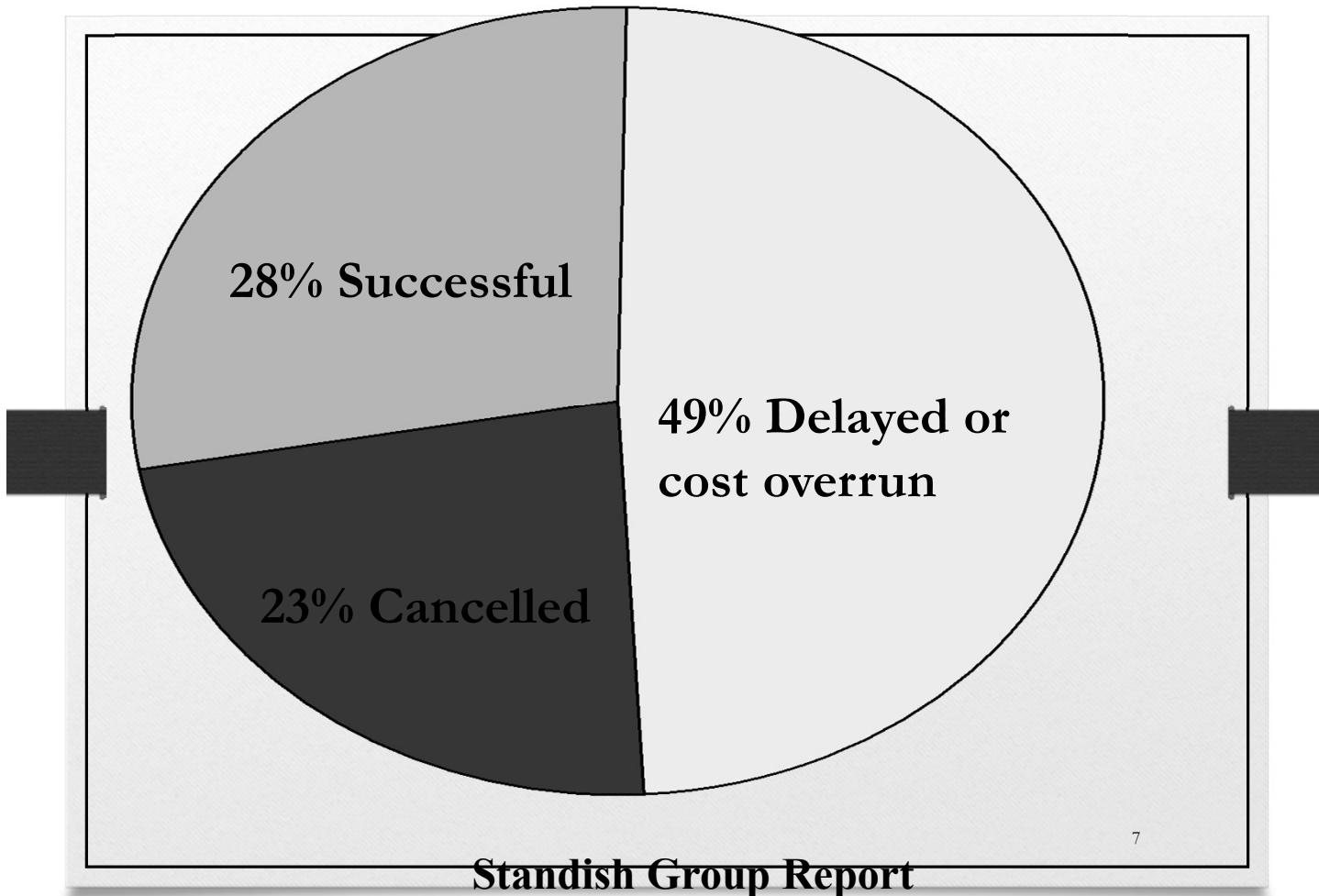
- It is often the case that software products:
 - Fail to meet user requirements.
 - Expensive.
 - Difficult to alter, debug, and enhance.
 - Often delivered late.
 - Use resources non-optimally.

Software Crisis (cont.)



~~hen why not have entirely hardware systems?...~~

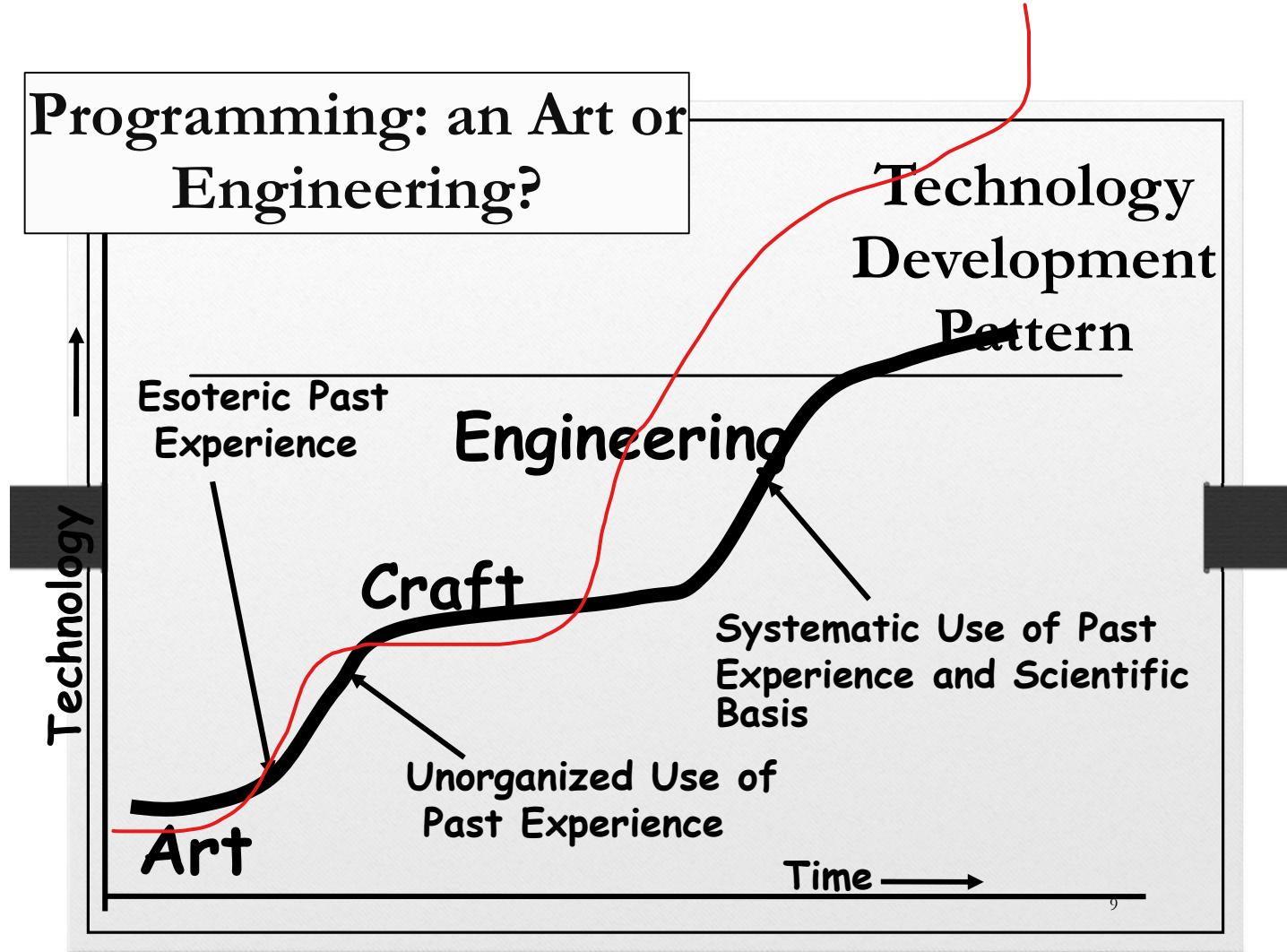
- A virtue of software:
 - Relatively easy and faster to develop and to change...
 - Consumes no space, weight, or power...
 - Otherwise all might as well be hardware.
- The more is the complexity of software, the harder it is to change--why?
 - Further, the more the changes made to a program, the greater becomes its complexity.



Which Factors are Contributing to the Software Crisis?

- Larger problems,
- Poor project management
- **Lack of adequate training in software engineering,**
- Increasing skill shortage,
- Low productivity improvements.

Programming: an Art or Engineering?



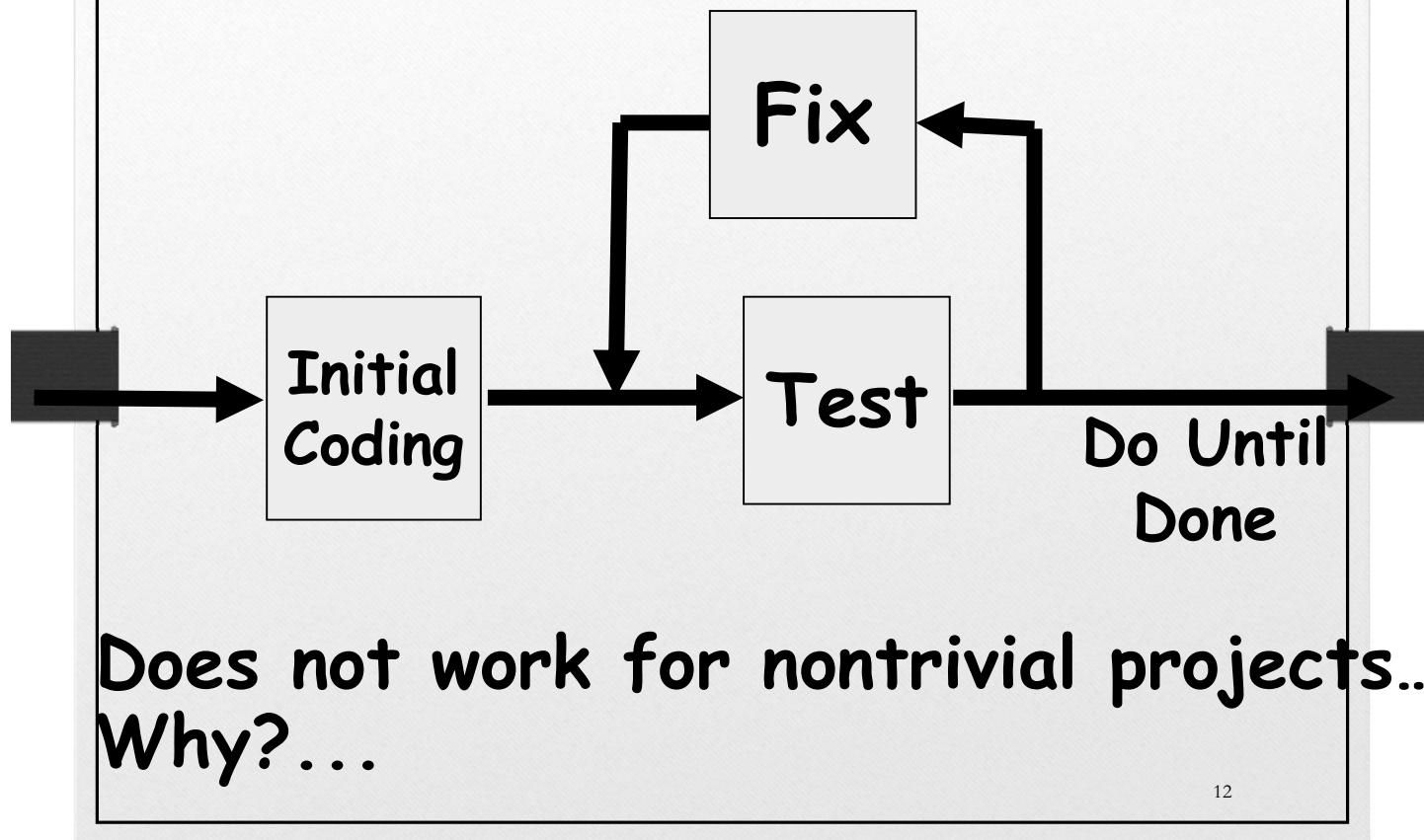
- Heavy use of past experience:
 - Past experience is systematically arranged.
- Theoretical basis and quantitative techniques provided.
- Many are just thumb rules.
- Tradeoff between alternatives.
- Pragmatic approach to cost-effectiveness.

Programming an Art
or Engineering?

What is Exploratory Software Development?

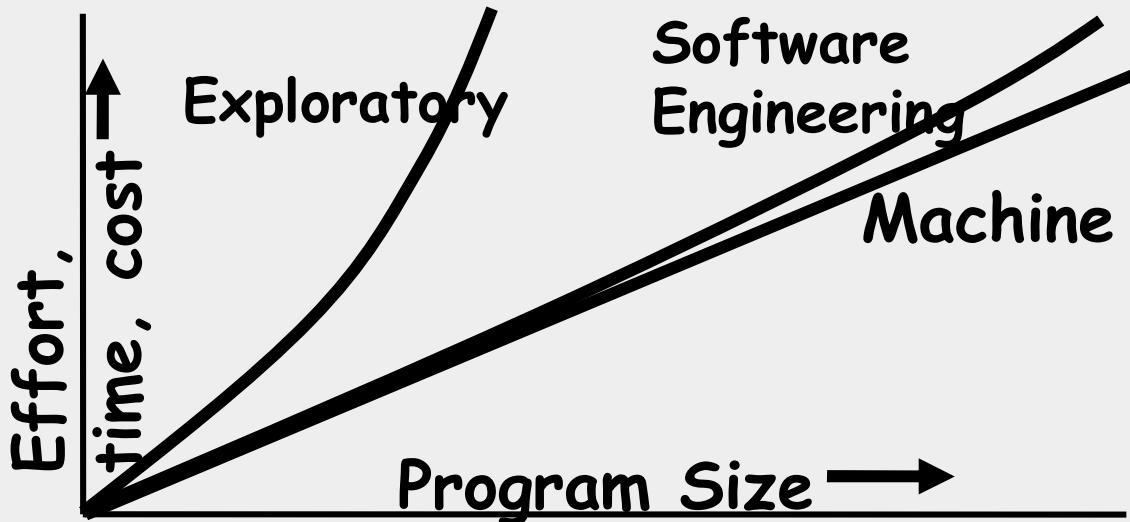
- Early programmers used **exploratory** (also called build and fix) style.
 - A 'dirty' program is quickly developed.
 - The bugs are fixed as and when they are noticed.
 - Similar to how a junior student develops programs...

Exploratory Style



What is Wrong with the Exploratory Style?

- Can successfully be used for developing only very small (toy) programs.



What is Wrong with the Exploratory Style?

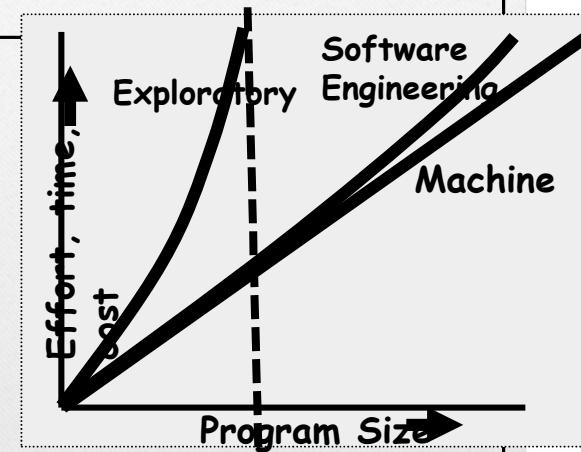
Cont...

- Besides the exponential growth of effort, cost, and time with problem size:
 - Exploratory style usually results in unmaintainable code.
 - **It becomes very difficult to use the exploratory style in team development environments...**

What is Wrong with the Exploratory Style? Cont...

- Why does the effort required to develop a software grow exponentially with size?

- Why does the approach completely breaks down when the size of software becomes large?



An Interpretation Based on Human Cognition Mechanism

- Human memory can be thought to be made up of two distinct parts [Miller 56]:
 - Short term memory and
 - Long term memory.

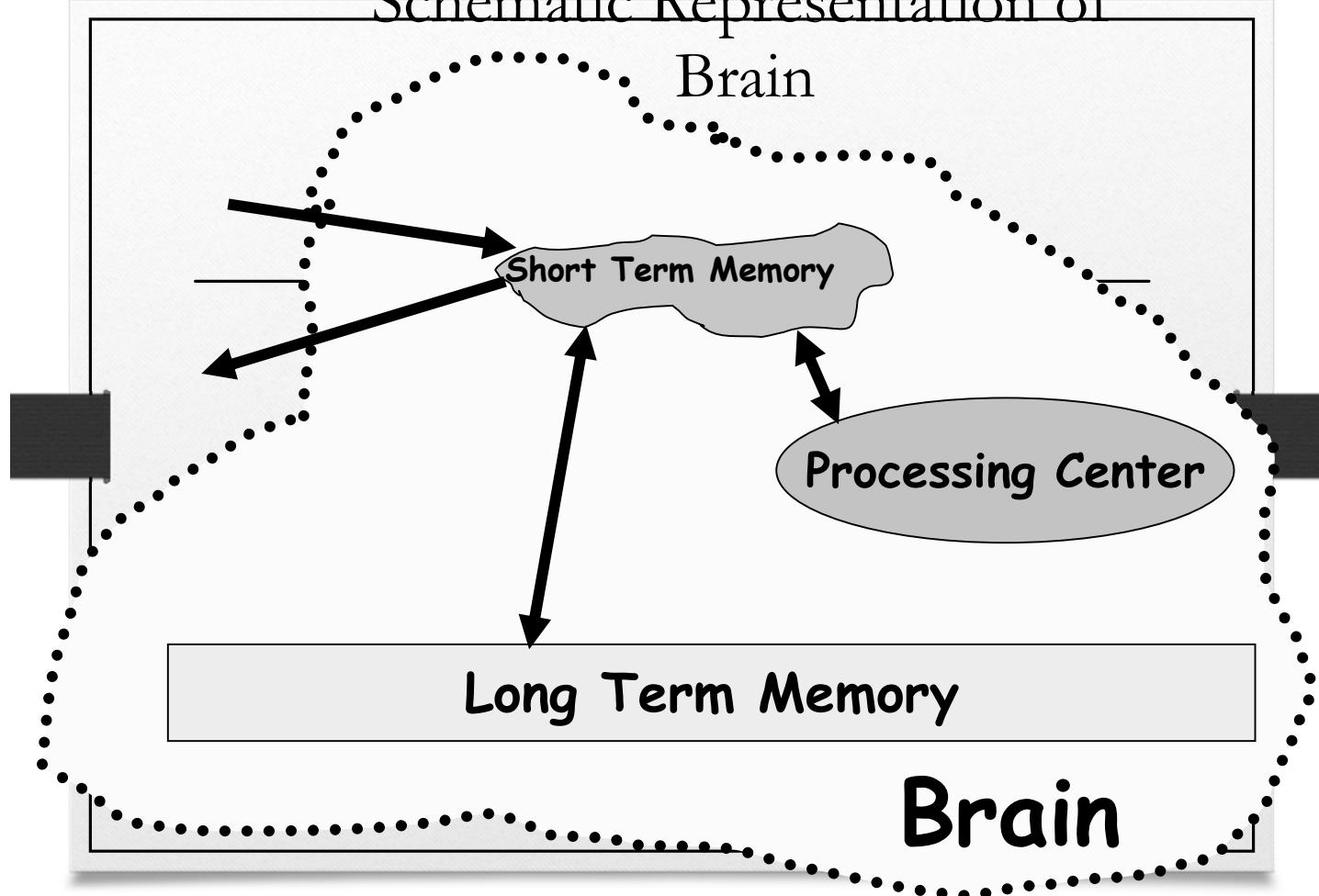
Human Cognition Mechanism

- Suppose I ask: “**It is 10:10AM now, how many hours are remaining today?**”

- 10AM would be stored in the short-term memory.
- “A day is 24 hours long.” would be fetched from the long term memory into short term memory.
- The mental manipulation unit would compute the difference (24-10).



Schematic Representation of



Short Term Memory

- An item stored in the short term memory can get lost:
 - Either due to decay with time or
 - **Displacement by newer information.**
- This restricts the time for which an item is stored in short term memory:
 - Typically few tens of seconds.
 - However, an item can be retained longer in the short term memory by recycling.

- **An item is any set of related information.**
 - A character such as 'a' or a digit such as '5'.
 - A word, a sentence, a story, or even a picture.
- Each item normally occupies one place in memory.
- When you are able to relate several different items together (chunking):
 - The information that should normally occupy several places, takes only one place in memory.

What is an Item?

Chunking

- If I ask you to remember the number

110010101001

- It may prove very hard for you to understand and remember.
- But, the octal form of **6251 (110)(010)(101)(001)** would be easier.
- You have managed to create chunks of three items each.

- In many of our day-to-day experiences:
 - Short term memory is evident.
- Suppose, you look up a number from the telephone directory and start dialling it.
 - If you find the number is busy, you can dial the number again after a few seconds without having to look up the number from directory.
- But, after several days:
 - You may not remember the number at all
 - Would need to consult the directory again.

Evidence
of Short
Term
Memory

- If a person deals with seven or less number of items:

The Magical Number 7

- These would be accommodated in the short term memory.
 - So, he can easily understand it.
-
- As the number of new information increases beyond seven:
 - It becomes exceedingly difficult to understand it.

What is the Implication in Program Development?

- A small program having just a few variables:
 - Is within easy grasp of an individual.
- As the number of independent variables in the program increases:
 - It quickly exceeds the grasping power of an individual...
 - Requires an unduly large effort to master the problem.

Implication in Program Development

- Instead of a human, if a machine could be writing (generating) a program,
 - The slope of the curve would be linear.
- But, how does use of software engineering principles helps hold down the effort-size curve to be almost linear?
- **Software engineering principles extensively use techniques specifically targeted to overcome the human cognitive limitations.**

Which Principles are Deployed by Software Engineering Techniques to Overcome Human Cognitive Limitations?

- Two important principles are profusely used:
 - **Abstraction**
 - **Decomposition**

Two Fundamental Techniques to Handle Complexity

What is Abstraction?

- Simplify a problem by omitting unnecessary details.
- Focus attention on only one aspect of the problem and ignore other aspects and irrelevant details.
- Also called model building.

Abstraction Example

- Suppose you are asked to develop an overall understanding of some country.
- Would you:
 - Meet all the citizens of the country, visit every house, and examine every tree of the country?
 - You would possibly refer to various types of maps for that country only.

You would study an Abstraction...

The image displays two maps of India side-by-side, each with a legend and a scale bar.

Left Map:

- Legend:**
 - International Boundary
 - State or Union Territory Boundary
 - Road
 - River
 - National Capital (star)
 - State or Union Territory Capital (star)
 - City or Town (dot)
- Scales:** 0-200 KM, 0-400 Miles
- Copyright:** © 2007 Geolup.com

Right Map:

- Legend:** None present
- Scales:** 80°E, 90°E
- Copyright:** Copyright © 2009 www.mapsofindia.com Last Updated on 21st Dec 2009

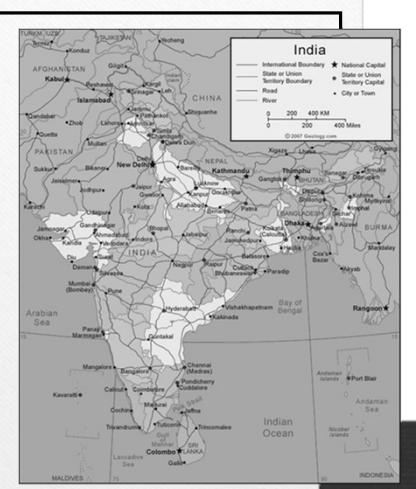
Both Maps Show:

- International boundaries with neighboring countries: TURKMENISTAN, AFGHANISTAN, PAKISTAN, NEPAL, BHUTAN, CHINA, and MYANMAR.
- Major cities labeled across the map.
- Major rivers labeled across the map.
- State or union territory boundaries.
- Road networks.

- A map is:
 - An abstract representation of a country.
 - Various types of maps (abstractions) possible.

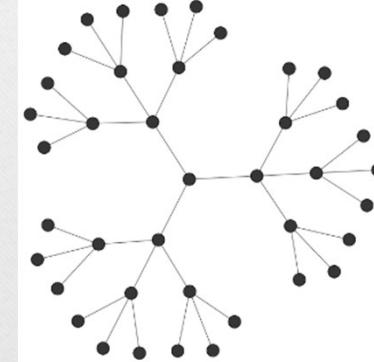
Does every Problem have a single Abstraction?

- Several abstractions of the same problem can be created:
 - Focus on some specific aspect and ignore the rest.
 - Different types of models help understand different aspects of the problem.



Abstractions of Complex Problems

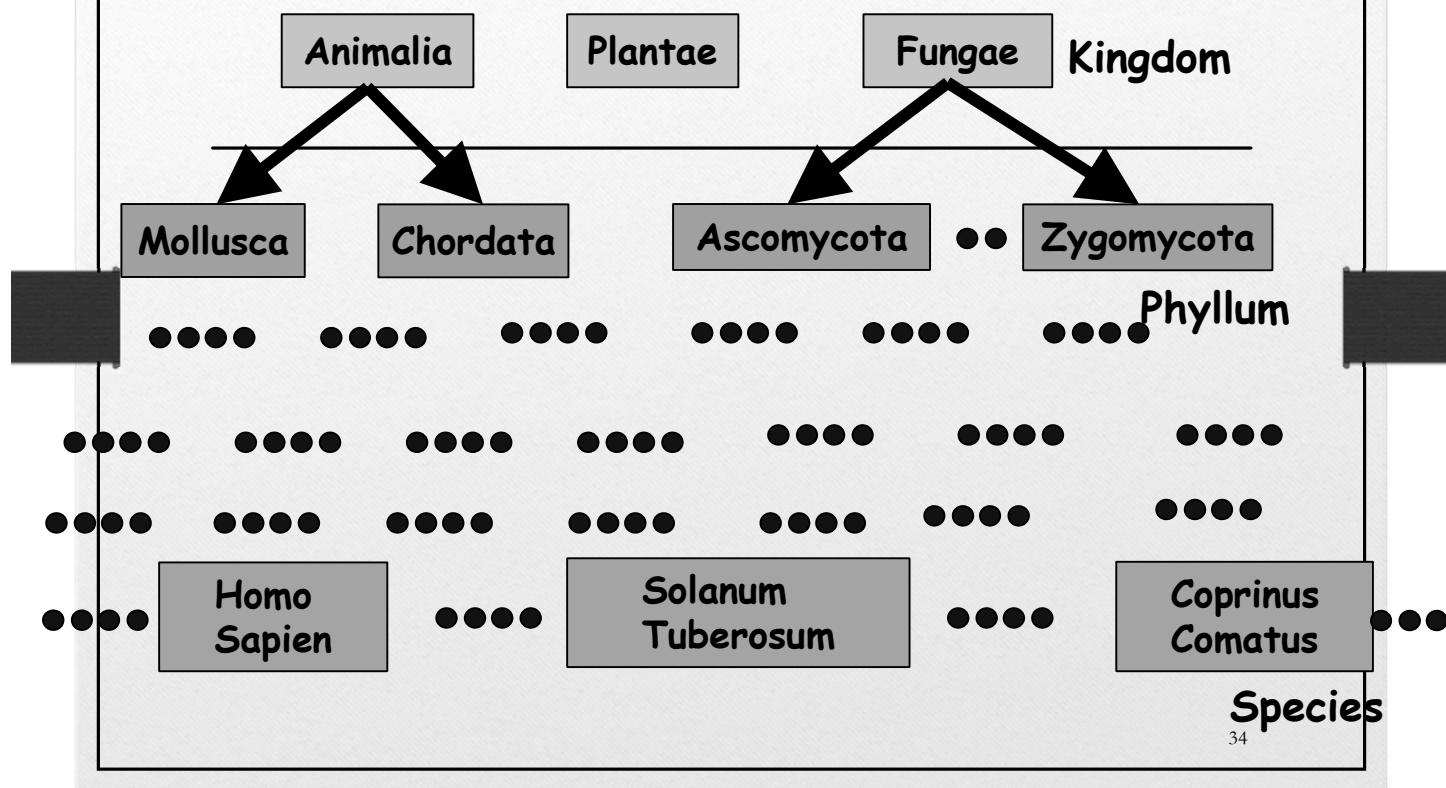
- For complex problems:
 - A single level of abstraction is inadequate.
 - A hierarchy of abstractions may have to be constructed.
- Hierarchy of models:
 - A model in one layer is an abstraction of the lower layer model.
 - An implementation of the model at the higher layer.³²



Abstraction of Complex Problems -- An Example

- Suppose you are asked to understand all life forms that inhabit the earth.
- Would you start examining each living organism?
 - You will almost never complete it.
 - Also, get thoroughly confused.
- **Solution: Try to build an abstraction hierarchy.**

Living Organisms



Quiz

- What is a model?
- Why develop a model? That is, how does constructing a model help?
- Give some examples of models.

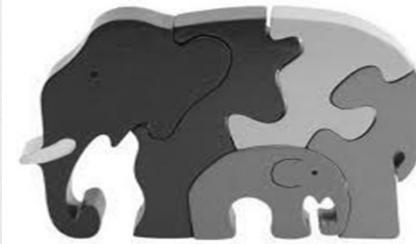
Decomposition

- Decompose a problem into many small independent parts.
 - The small parts are then taken up one by one and solved separately.
 - **The idea is that each small part would be easy to grasp and therefore can be easily solved.**
 - **The full problem is solved when all the parts are solved.**



Decomposition

- A popular example of decomposition principle:
 - Try to break a bunch of sticks tied together versus breaking them individually.
- Any arbitrary decomposition of a problem may not help.
 - The decomposed parts must be more or less independent of each other.

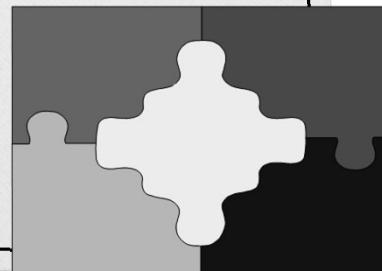


Decomposition: Another Example

- Example use of decomposition principle:
 - You understand a book better when the contents are organized into independent chapters.
 - Compared to when everything is mixed up.

Why Study Software Engineering? (1)

- To acquire skills to develop large programs.
- Handling exponential growth in complexity with size.
- Systematic techniques based on abstraction (modelling) and decomposition.



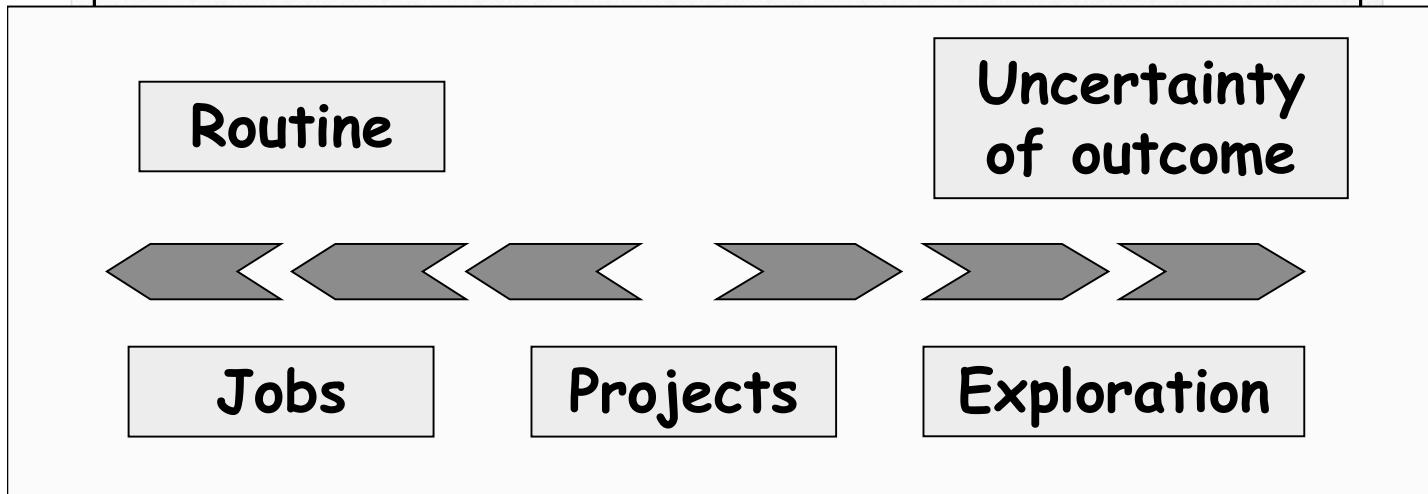
Why Study Software Engineering? (2)

- Learn systematic techniques of:
 - Specification, design, user interface development, testing, project management, maintenance, etc.
 - Appreciate issues that arise in team development.

Why Study Software Engineering? (3)

- To acquire skills to be a better
programmer:
 - Higher Productivity
 - Better Quality Programs

~~Jobs versus Projects~~



Jobs – repetition of very well-defined and well understood tasks with very little uncertainty

Exploration – The outcome is very uncertain, e.g. finding a cure for cancer.

Projects – in the middle! Has challenge as well as routine ...

Types of Software Projects

- Two types of software projects:
 - **Products (Generic software)**
 - **Services (custom software)**
- Total business – Several Trillions of US \$
- Half in products and half services
- **Services segment is growing fast!**

Types of Software

Packaged software –
prewritten software available for purchase

Horizontal market software—meets needs of many companies

Custom software –
software developed at some user's requests-Usually developer tailors some generic solution

Vertical market software—designed for particular industry

Thank You!!

Introduction

Continued....

Types of Software Projects

- Software product development projects
- Software services projects

Software Services

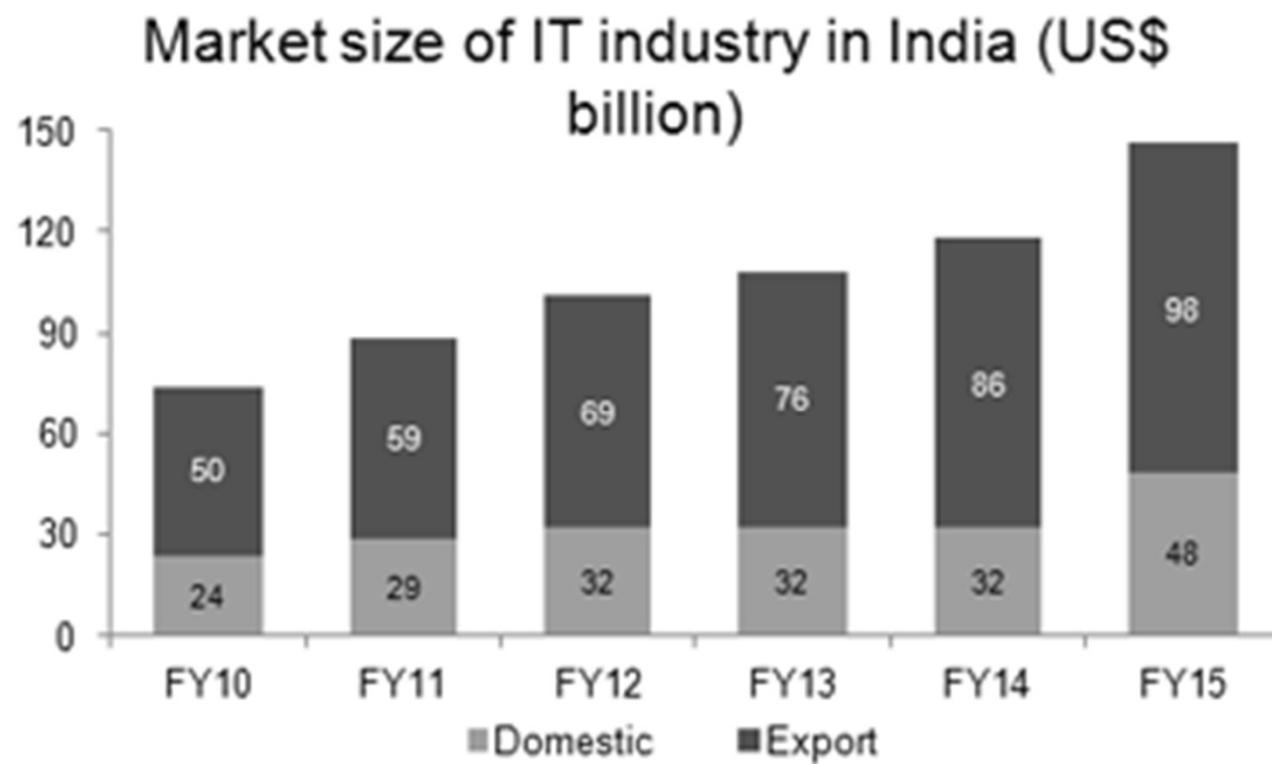
- Software service is an umbrella term, includes:
 - Software customization
 - Software maintenance
 - Software testing
 - Also contract programmers (CP) carrying out coding or any other assigned activities.



Factors responsible for accelerated growth of services...

- Now lots of code is available in a company:
 - New software can be developed by modifying the closest.
- Speed of Conducting Business has increased tremendously:
 - Requires shortening of project duration

Contribution of the IT sector to India's GDP rose to approximately 9.5% in 2015 from 1.2% in 98



Source: Nasscom, TechSci Research; Note: E - Estimates

Scenario of Indian Software Companies

- **Indian companies have largely focused on the services segment --- Why?**

Few Changes in Software Project Characteristics over Last 40 Years

- 40 years back, very few software existed
 - Every project started from scratch
 - Projects were multi year long
- The programming languages that were used earlier hardly provided any scope for reuse:
 - FORTRAN, PASCAL, COBOL, BASIC
- No application was GUI-based:
 - Mostly command selection from displayed text menu items.

Traditional versus Modern Projects

- Projects are increasingly becoming services:
 - Either tailor some existing software or reuse pre-built libraries.
- Facilitate and accommodate client feedbacks
- Facilitate customer participation in project development work
- Incremental software delivery with evolving functionalities.
- **No software is being developed from scratch --- Significant reuse is being made...**

Computer Systems Engineering

- **Many products require development of software as well as specific hardware to run it:**
 - a coffee vending machine,
 - a robotic toy,
 - A new health band product, etc.
- Computer systems engineering:
 - encompasses software engineering.

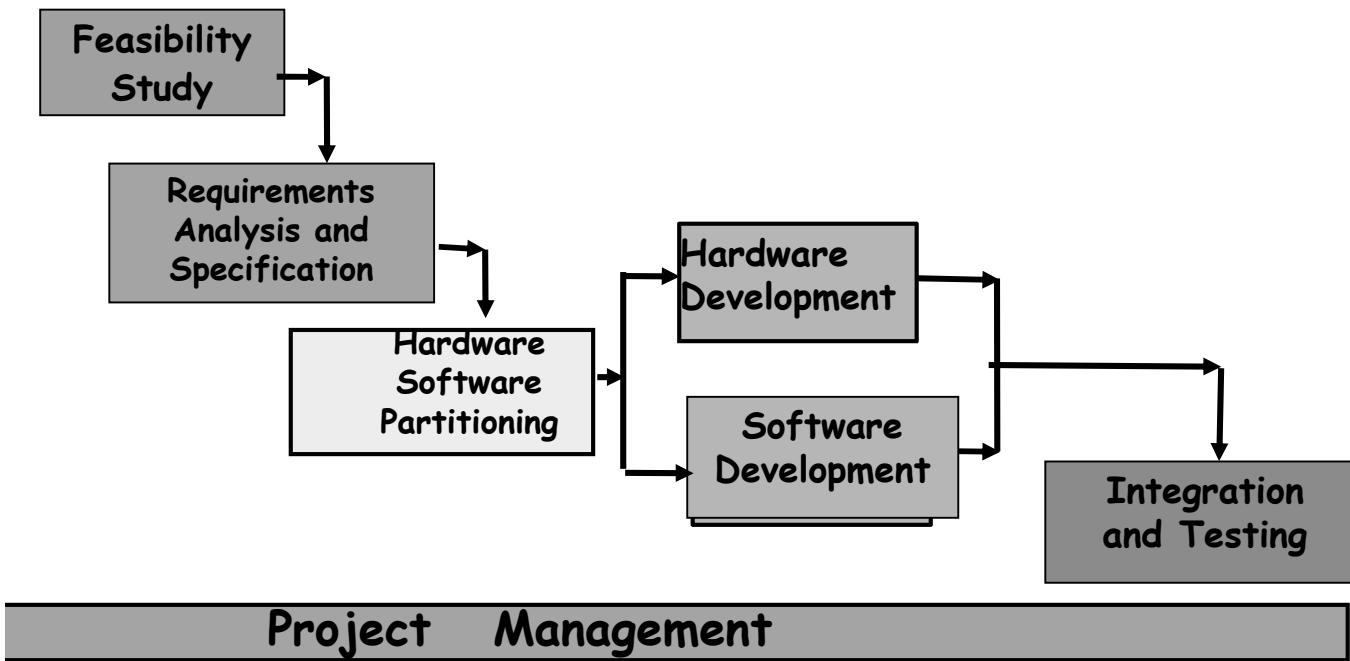
Computer Systems Engineering

- The high-level problem:
 - Deciding which tasks are to be solved by software.
 - Which ones by hardware.

Computer Systems Engineering (CONT.)

- Typically, hardware and software are developed together:
 - Hardware simulator is used during software development.
- Integration of hardware and software.
- Final system testing

Computer Systems Engineering (CONT.)



Emergence of Software Engineering Techniques

Emergence of Software Engineering Techniques

- Early Computer Programming (1950s):
 - Programs were being written in assembly language...
 - Sizes limited to about a few hundreds of lines of assembly code...

Early Computer Programming (50s)

- Every programmer developed his/her own style of writing programs:
 - According to his intuition (called exploratory or build-and-fix programming) .

High-Level Language Programming (Early 60s)

- High-level languages such as FORTRAN, ALGOL, and COBOL were introduced:
 - This reduced software development efforts greatly.
 - Why reduces?

High-Level Language Programming (Early 60s)

- **Software development style was still exploratory.**
 - Typical program sizes were limited to a few thousands of lines of source code.

Control Flow-Based Design (late 60s)

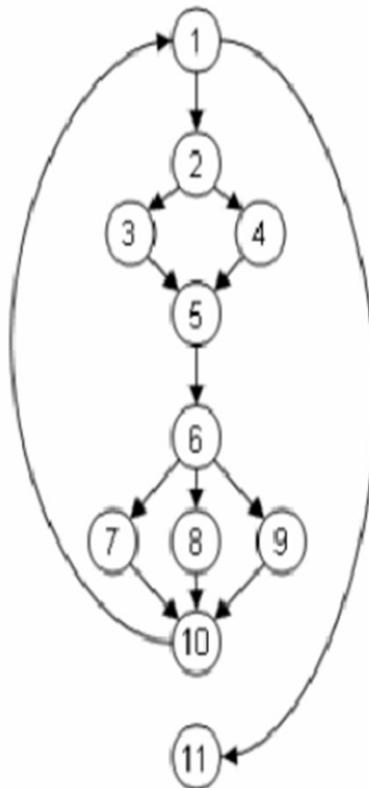
- Size and complexity of programs increased further:
 - Exploratory programming style proved to be insufficient.
- Programmers found:
 - Very difficult to write cost-effective and correct programs.

Control Flow-Based Design (late 60s)

- Programmers found it very difficult:
 - To understand and maintain programs written by others.
- To cope up with this problem, experienced programmers advised---"Pay particular attention to the design of the program's control structure."

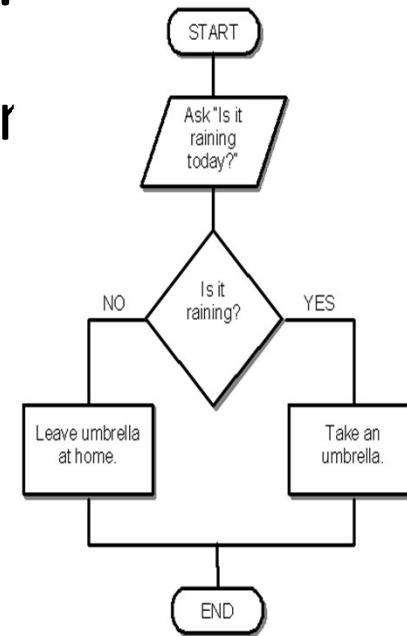
Control Flow-Based Design (late 60s)

- What is a program's control structure?
 - Sequence in which a program's instructions are executed.
- To help design programs having good control structure:
 - Flow charting technique was developed.



Control Flow-Based Design (late 60s)

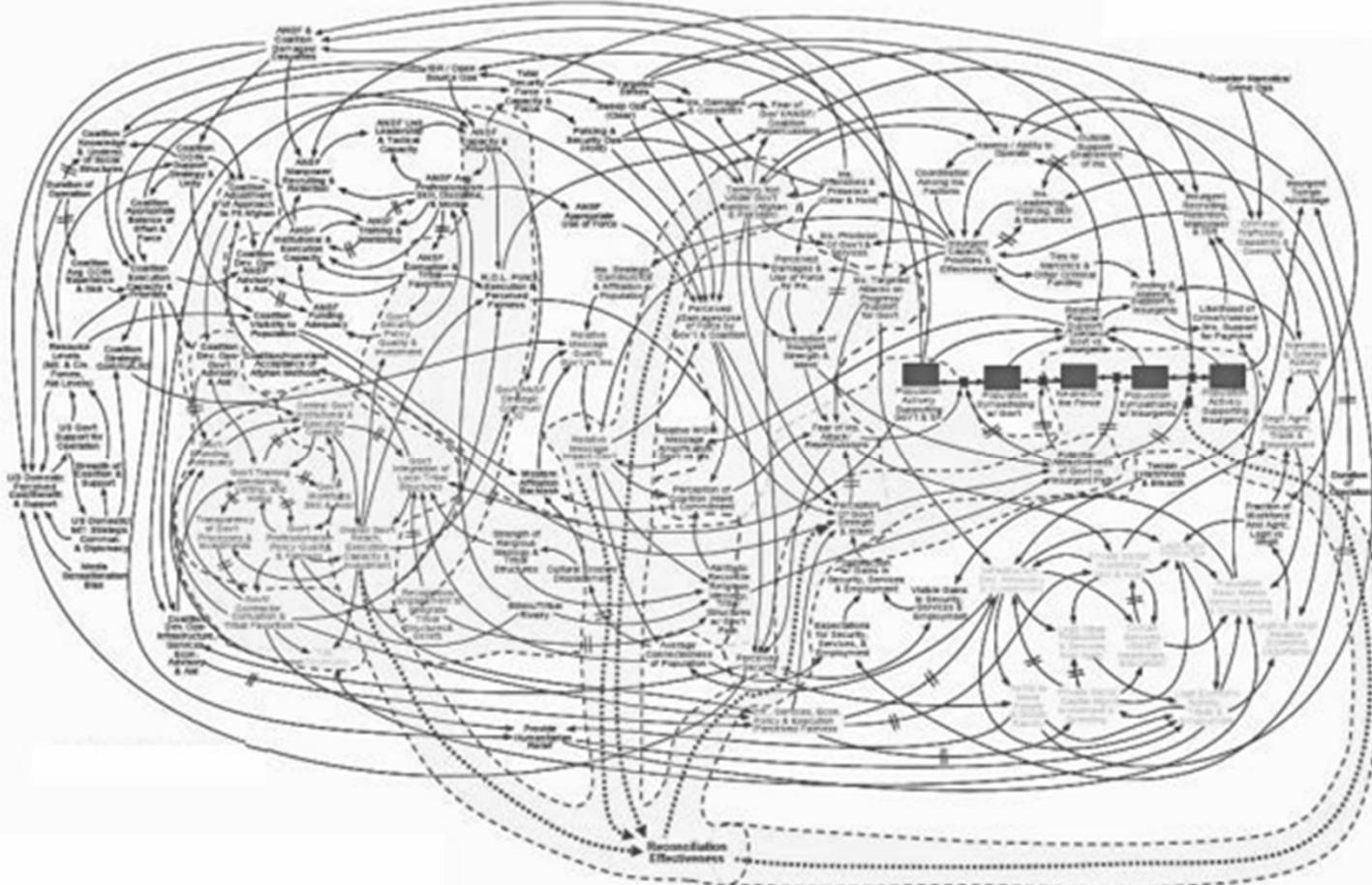
- Using flow charting technique:
 - One can represent and design program's control structure.
 - When asked to understand a program:
 - One would mentally trace the program's execution sequence.



Control Flow-Based Design

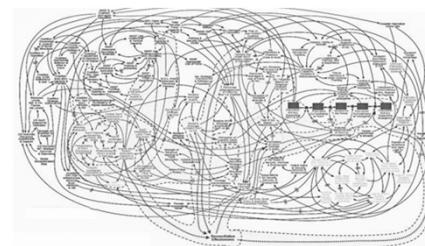
- A program having a messy flow chart representation:
 - Difficult to understand and debug.

~~Spaghetti Code Structure~~



Control Flow-Based Design (Late 60s)

- What causes program complexity?
 - GO TO statements makes control structure of a program messy.
 - GO TO statements alter the flow of control arbitrarily.
 - The need to restrict use of GO TO statements was recognized.



Control Flow-Based Design

(Late 60s)

- Many programmers had extensively used assembly languages.
 - JUMP instructions are frequently used for program branching in assembly languages.
 - Programmers considered use of GO TO statements inevitable.

```
addi $a0, $0, 1
j next
next:
j skip1
add $a0, $a0, $a0
skip1:
j skip2
add $a0, $a0, $a0
add $a0, $a0, $a0
skip2:
j skip3
loop:
add $a0, $a0, $a0
add $a0, $a0, $a0
add $a0, $a0, $a0
skip3:
j loop
```

Control-flow Based Design (Late 60s)

- At that time, Dijkstra published his article:
 - “Goto Statement Considered Harmful”
Comm. of ACM, 1969.
- Many programmers were unhappy to read his article.

Control Flow-Based Design (Late 60s)

- Some programmers published several counter articles:
 - Highlighted the advantages and inevitability of GO TO statements.

Control Flow-Based Design (Late 60s)

- It soon was conclusively proved:
 - Only three programming constructs are sufficient to express any programming logic:
 - **sequence** (`a=0;b=5;`)
 - **selection** (`if(c==true) k=5 else m=5;`)
 - **iteration** (`while(k>0) k=j-k;`)

Control-flow Based Design (Late 60s)

- Everyone accepted:
 - It is possible to solve any programming problem without using GO TO statements.
 - This formed the basis of **Structured Programming methodology.**

Structured Programming

- A program is called structured:
 - When it uses only the following types of constructs:
 - sequence,
 - selection,
 - iteration
 - Consists of modules.

Structured Programs

- Sometimes, violations to structured programming are permitted:
 - Due to practical considerations such as:
 - Premature loop exit (break) or for exception handling.

Advantages of Structured programming

- Structured programs are:
 - Easier to read and understand,
 - Easier to maintain,
 - Require less effort and time for development.
 - Less buggy

Structured Programming

- Research experience shows:
 - Programmers commit less number of errors:
 - While using structured if-then-else and do-while statements.
 - Compared to test-and-branch (GOTO) constructs.

Data Structure-Oriented Design (Early 70s)

- As program sizes increased further, soon it was discovered:
 - It is important to pay more attention to the design of data structures of a program**
 - Than to the design of its control structure.

Data Structure-Oriented Design (Early 70s)

- Techniques which emphasize designing the data structure:
 - Derive program structure from it:
 - Are called **data structure-oriented design techniques**.

Data Structure Oriented Design (Early 70s)

- An example of data structure-oriented design technique:
 - Jackson's Structured Programming(JSP) methodology
- Developed by Michael Jackson in 1970s.

Data Structure Oriented Design (Early 70s)

- **JSP technique:**
 - Program code structure should correspond to the data structure.

A Data Structure Oriented Design (Early 70s)

●JSP methodology:

- A program's data structures are first designed using notations for

●sequence, selection, and iteration.

- The data structure design is then used :
 - To derive the program structure.

Data Structure Oriented Design (Early 70s)

- Several other data structure-oriented Methodologies also exist:
 - e.g., Warnier-Orr Methodology.

Data Flow-Oriented Design (Late 70s)

- Data flow-oriented techniques advocate:
 - The data items input to a system must first be identified,
 - Processing required on the data items to produce the required outputs should be determined.

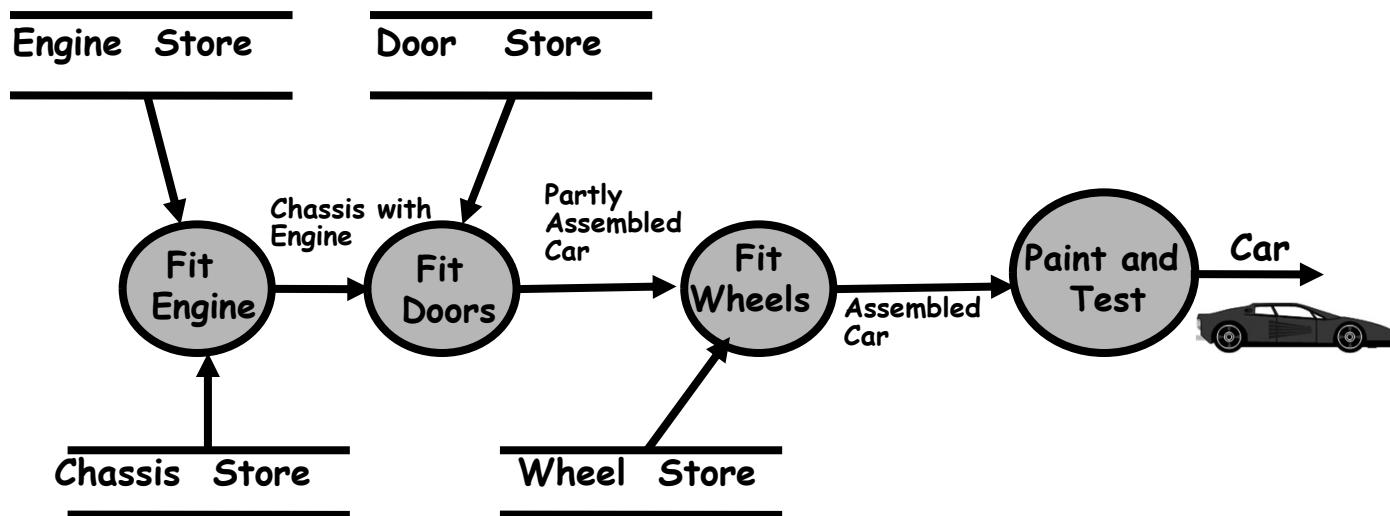
Data Flow-Oriented Design (Late 70s)

- Data flow technique identifies:
 - Different processing stations (functions) in a system.
 - The items (data) that flow between processing stations.

Data Flow-Oriented Design (Late 70s)

- Data flow technique is a generic technique:
 - Can be used to model the working of any system.
 - not just software systems.
- A major advantage of the data flow technique is its simplicity.

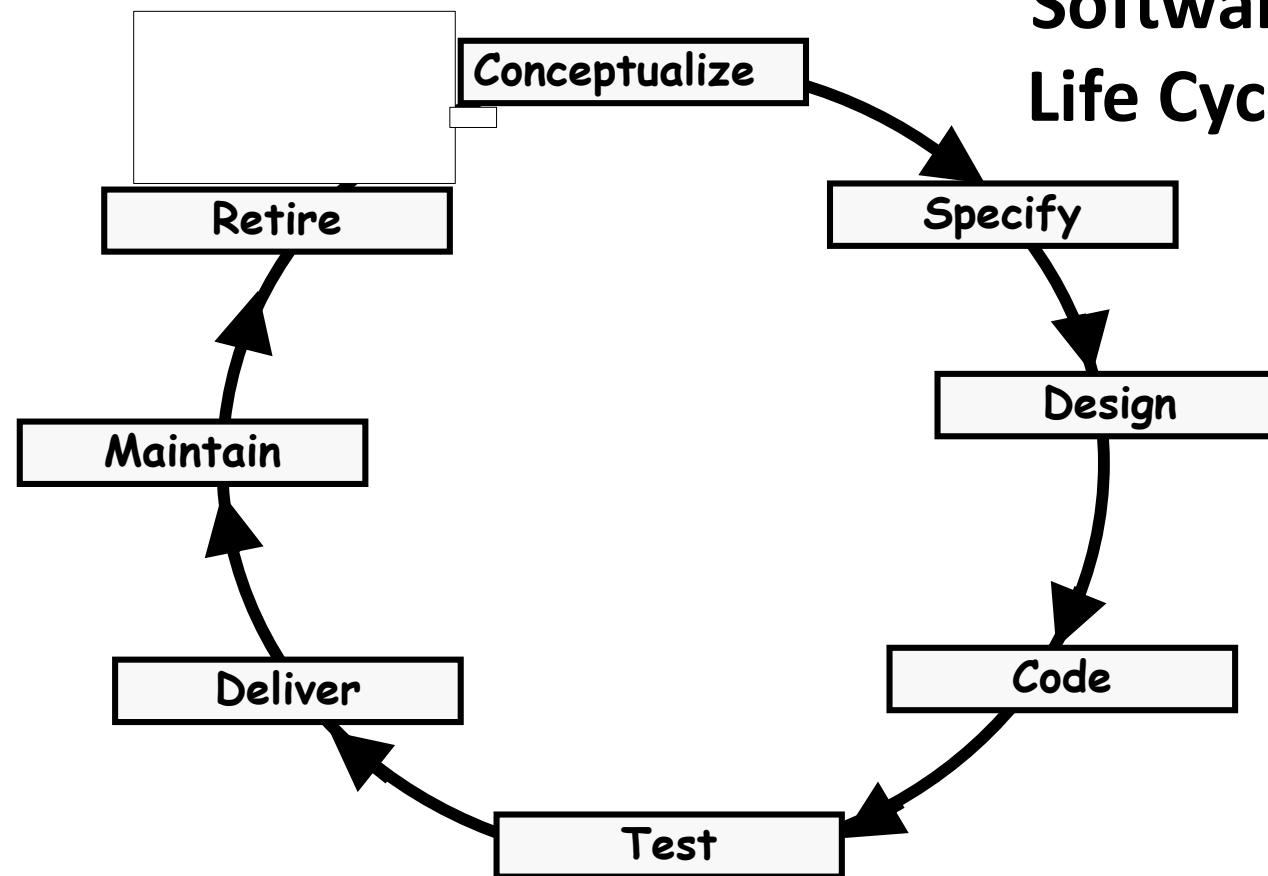
Data Flow Model of a Car Assembly Unit



Thank You!!

Life Cycle Models

Software Life Cycle



Life Cycle Model

- A software life cycle model (also process model or SDLC):
 - A ~~descriptive and diagrammatic~~ model of software life cycle:
 - Identifies all the activities undertaken during product development,
 - Establishes a precedence ordering among the different activities,
 - Divides life cycle into phases.

Life Cycle Model (CONT.)

- Each life cycle phase consists of several activities.
 - For example, the design stage might consist of:
 - structured analysis
 - structured design
 - Design review

Why Model Life Cycle?

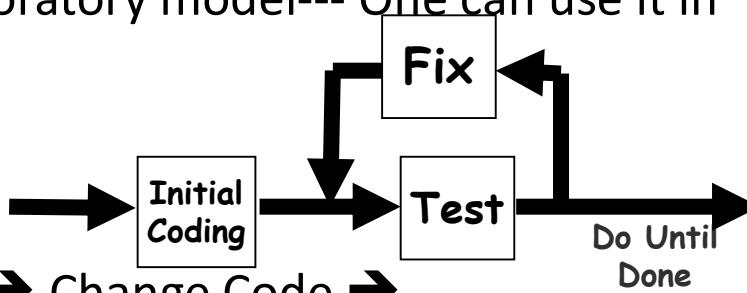
- A graphical and written description:
 - Helps common understanding of activities among the software developers.
 - Helps to identify inconsistencies, redundancies, and omissions in the development process.
 - Helps in tailoring a process model for specific projects.

Life Cycle Model (CONT.)

- The development team must identify a suitable life cycle model:
 - and then adhere to it.
 - Primary advantage of adhering to a life cycle model:
 - Helps development of software in a systematic and disciplined manner.

Life Cycle Model (CONT.)

- When a program is developed by a single programmer ---
 - The problem is within the grasp of an individual.
 - He has the freedom to decide his exact steps and still succeed --- called Exploratory model--- One can use it in many ways
 - Code → Test → Design
 - Code → Design → Test → Change Code →
 - Specify → code → Design → Test → etc.



Life Cycle Model (CONT.)

- When software is being developed by a team:
 - There must be a precise understanding among team members as to when to do what,
 - Otherwise, it would lead to chaos and project failure.

Life Cycle Model (CONT.)

- A software project will never succeed if:
 - one engineer starts writing code,
 - another concentrates on writing the test document first,
 - yet another engineer first defines the file structure
 - another defines the I/O for his portion first.

Phase Entry and Exit Criteria

- A life cycle model:
 - defines entry and exit criteria for every phase.
 - A phase is considered to be complete:
 - only when all its exit criteria are satisfied.



10

Life Cycle Model (CONT.)

- What is the phase exit criteria for the software requirements specification phase?
 - Software Requirements Specification (SRS) document is complete, reviewed, and approved by the customer.
- A phase can start:
 - Only if its phase-entry criteria have been satisfied.

Life Cycle Model: Milestones

- Milestones help software project managers:
 - Track the progress of the project.
 - Phase entry and exit are important milestones.



Life Cycle and Project Management

- When a life cycle model is followed:
 - The project manager can at any time fairly accurately tell,
 - At which stage (e.g., design, code, test, etc.) the project is.

Project Management Without Life Cycle Model

- It becomes very difficult to track the progress of the project.
 - The project manager would have to depend on the guesses of the team members.
- This usually leads to a problem:
 - known as the **99% complete syndrome.**

Project Deliverables: Myth and Reality

Myth:

The only deliverable for a successful project is the working program.

Reality:

Documentation of all aspects of software development are needed to help in operation and maintenance.

- Many life cycle models have been proposed.
- We confine our attention to only a few commonly used models.

– **Waterfall**

– **V model,**

– **Evolutionary,**

– **Prototyping**

– **Spiral model,**

– **Agile models**

Life Cycle Model (CONT.)

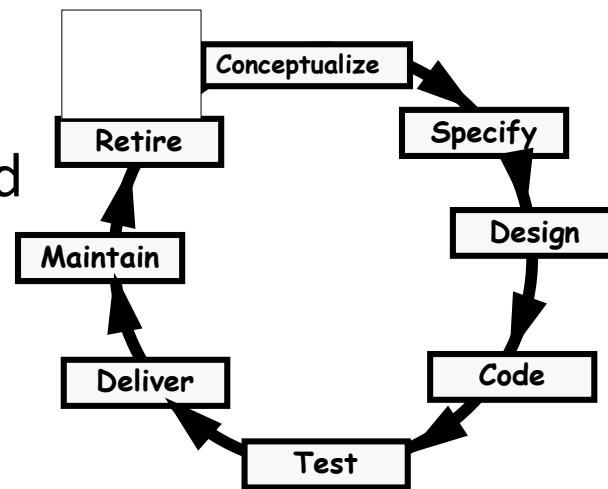
Traditional models

- Software life cycle (or software process):
 - Series of identifiable stages that a software product undergoes during its life time:
 - Feasibility study
 - Requirements analysis and specification,
 - Design,
 - Coding,
 - Testing
 - Maintenance.

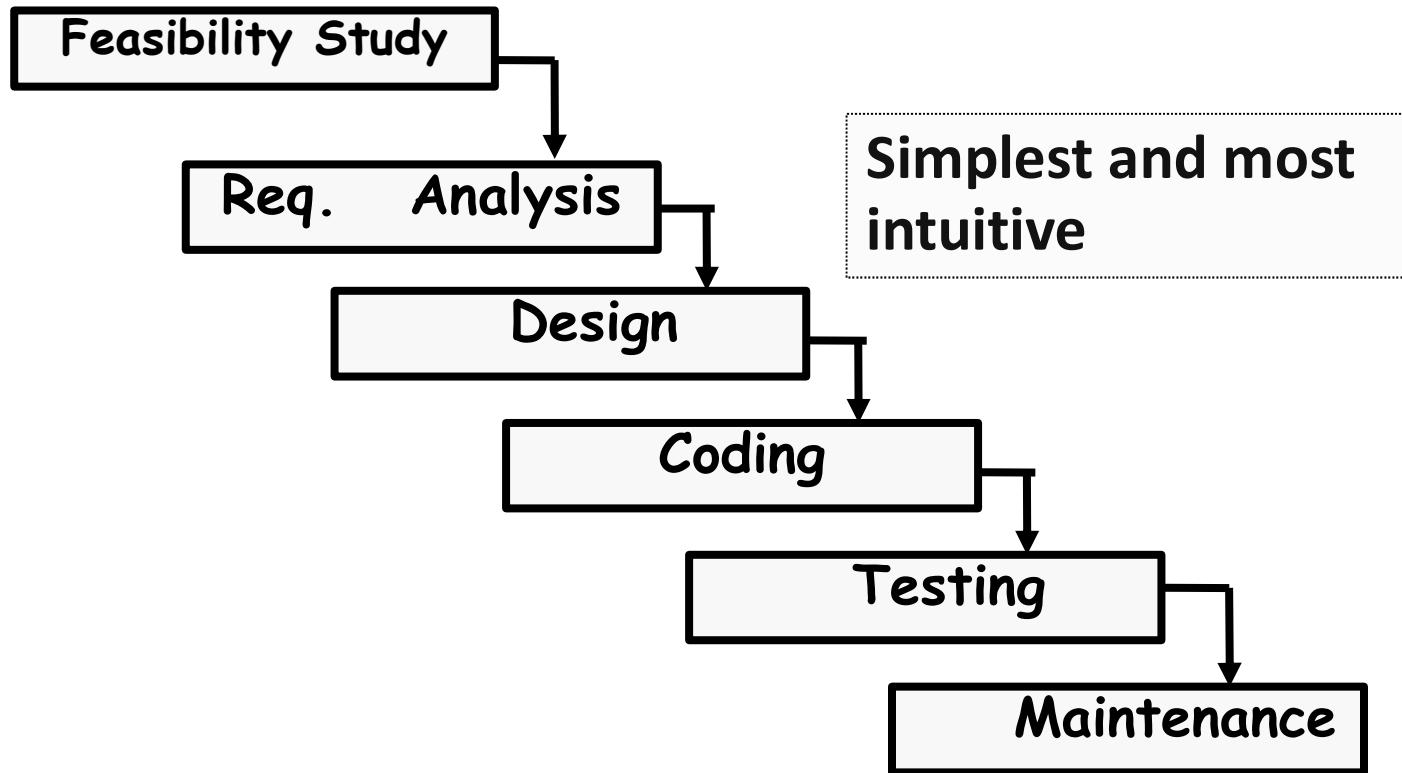
Software Life Cycle

Classical Waterfall Model

- Classical waterfall model divides life cycle into following phases:
 - Feasibility study,
 - Requirements analysis and specification,
 - Design,
 - Coding and unit testing,
 - Integration and system testing,
 - Maintenance.

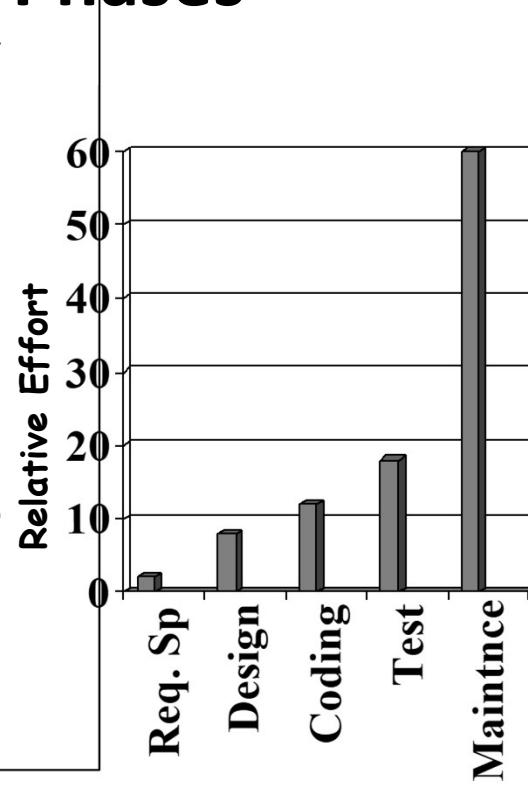


Classical Waterfall Model



Relative Effort for Phases

- Phases between feasibility study and testing
 - Called development phases.
- Among all life cycle phases
 - Maintenance phase consumes maximum effort.
- Among development phases,
 - Testing phase consumes the maximum effort.



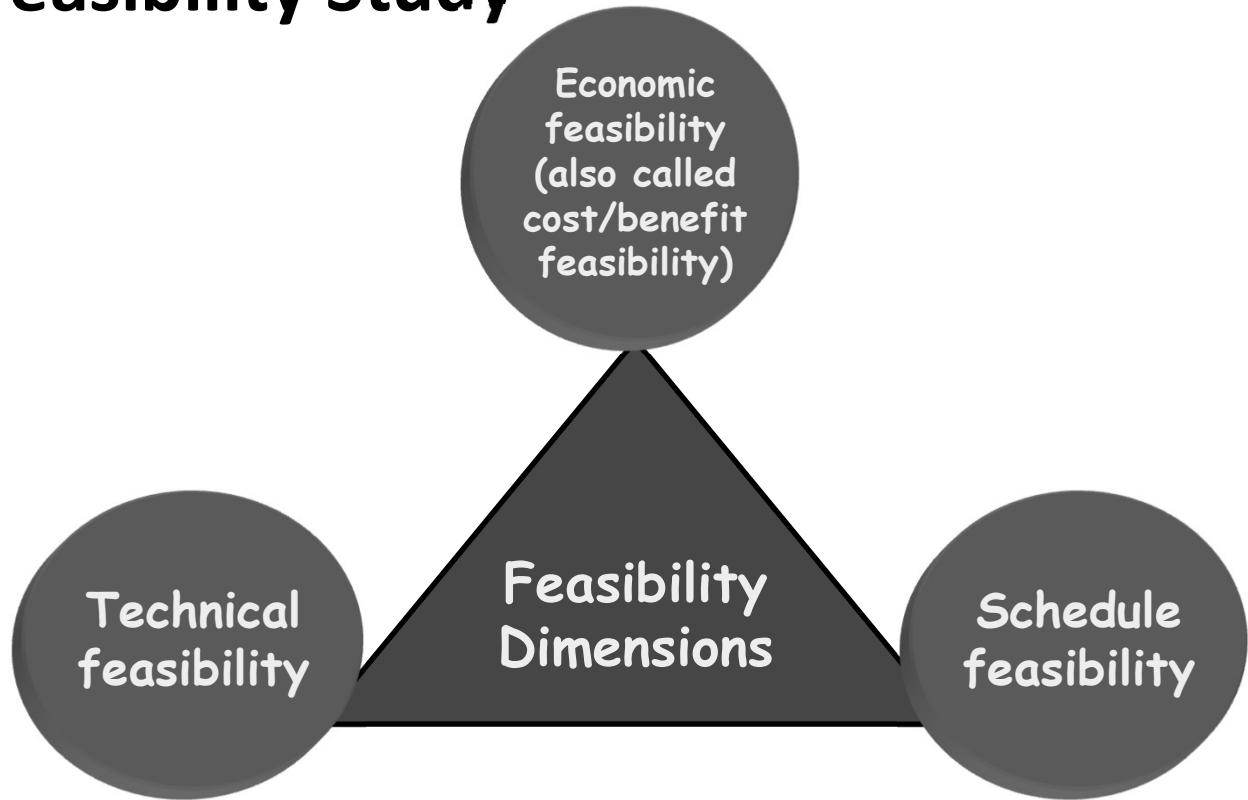
- Most organizations usually define:
 - Standards on the outputs (deliverables) produced at the end of every phase
 - Entry and exit criteria for every phase.
- They also prescribe methodologies for:
 - Specification,
 - Design,
 - Testing,
 - Project management, etc.

Process Model

Classical Waterfall Model (CONT.)

- The guidelines and methodologies of an organization:
 - Called the organization's software development methodology.
- Software development organizations:
 - Expect fresh engineers to master the organization's software development methodology.

Feasibility Study



- Main aim of feasibility study: determine whether developing the software is:
 - Financially worthwhile
 - Technically feasible.
- Roughly understand what customer wants:
 - Data which would be input to the system,
 - Processing needed on these data,
 - Output data to be produced by the system,
 - Various constraints on the behavior of the system.

Feasibility Study

First Step

- SPF Scheme for CFL
- CFL has a large number of employees, exceeding 50,000.
- Majority of these are casual labourers
- Mining being a risky profession:
 - Casualties are high
- Though there is a PF:
 - But settlement time is high
- There is a need of SPF:
 - For faster disbursement of benefits

Case Study

Feasibility: Case Study

- Manager visits main office, finds out the main functionalities required
- Visits mine site, finds out the data to be input
- Suggests alternate solutions
- Determines the best solution
- Presents to the CFL Officials
- Go/No-Go Decision

Activities During Feasibility Study

- Work out an overall understanding of the problem.
- Formulate different solution strategies.
- Examine alternate solution strategies in terms of:
 - resources required,
 - cost of development, and
 - development time.

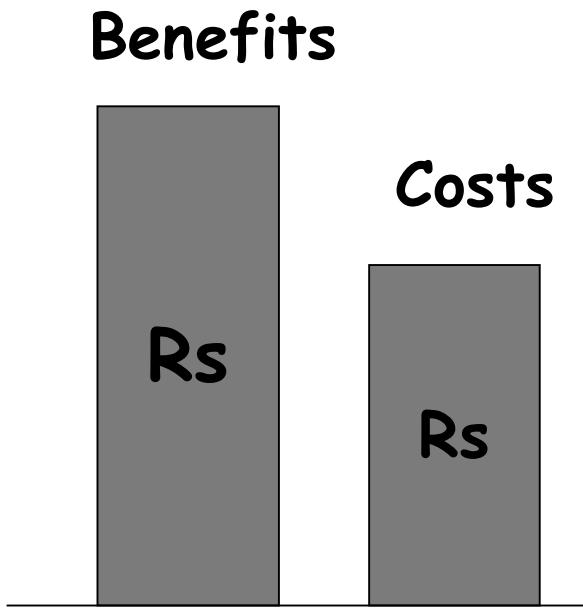
- Perform a cost/benefit analysis:
 - Determine which solution is the best.
 - May also find that none of the solutions is feasible due to:
 - high cost,
 - resource constraints,
 - technical reasons.

**Activities during
Feasibility Study**

Cost benefit analysis (CBA)

- Need to identify all costs --- these could be:
 - **Development costs**
 - **Set-up**
 - **Operational costs**
- Identify the value of benefits
- Check benefits are greater than costs

The business case



- Benefits of delivered project must outweigh costs
- Costs include:
 - Development
 - Operation
- Benefits:
 - Quantifiable
 - Non-quantifiable

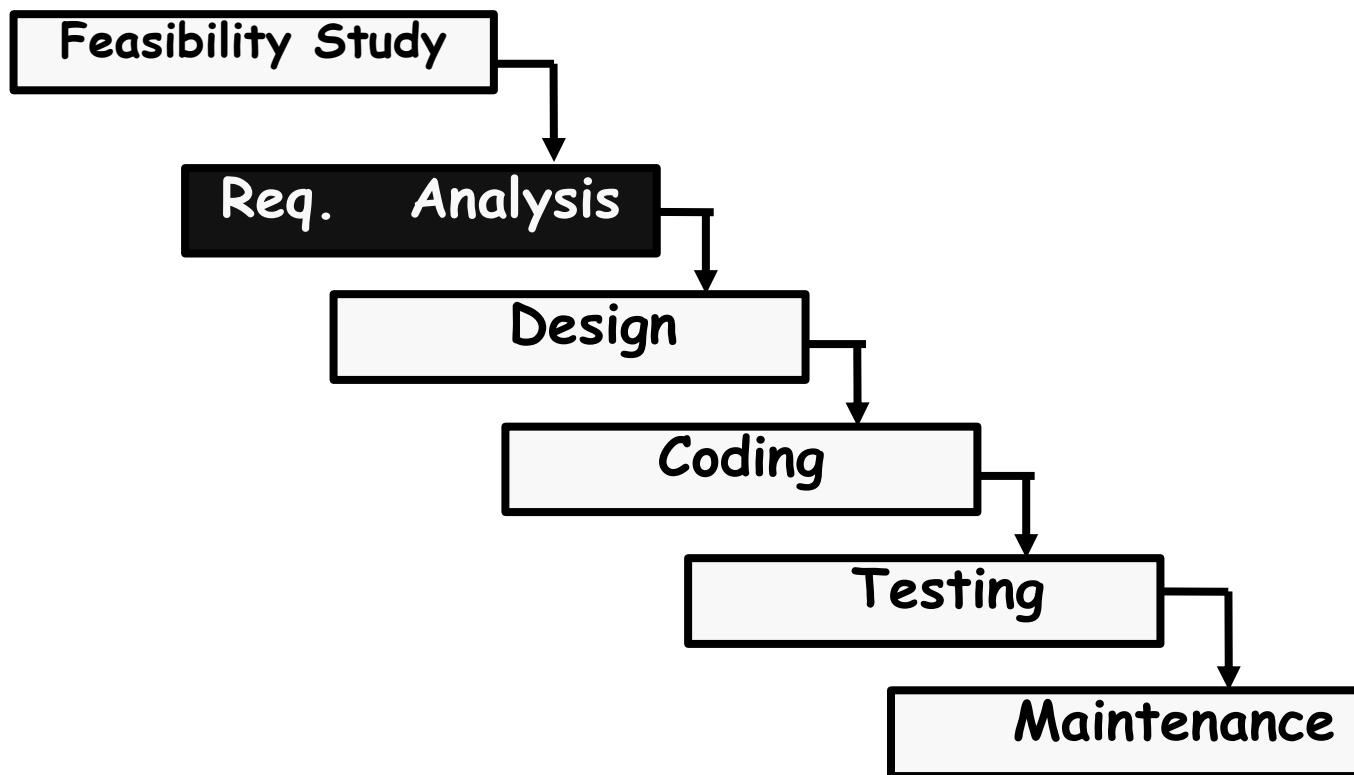
The business case

- Feasibility studies should help write a ‘business case’
- Should provide a justification for starting the project
- Should show that the benefits of the project will exceed:
 - **Various costs**
- Needs to take account of business risks

- 1. Executive summary**
- 2. Project background:**
 - The focus must be on what, exactly, the project is undertaking, and should not be confused with what might be a bigger picture.
- 3. Business opportunity**
 - What difference will it make?
 - What if we don't do it?
- 4. Costs**
 - Should include the cost of development, implementation, training, change management, and operations.
- 5. Benefits**
 - Benefits usually presented in terms of revenue generation and cost reductions.
- 6. Risks**
 - Identify risks.
 - Explain how these will be managed.

Writing an Effective Business Case

Classical Waterfall Model



Requirements Analysis and Specification

- Aim of this phase:
 - Understand the exact requirements of the customer,
 - Document them properly.
- Consists of two distinct activities:
 - Requirements gathering and analysis
 - Requirements specification.

Requirements Analysis and Specification

- Gather requirements data from the customer:
 - Analyze the collected data to understand what customer wants
- Remove requirements problems:
 - Inconsistencies
 - Anomalies
 - Incompleteness
- Organize into a Software Requirements Specification (SRS) document.

Requirements Gathering

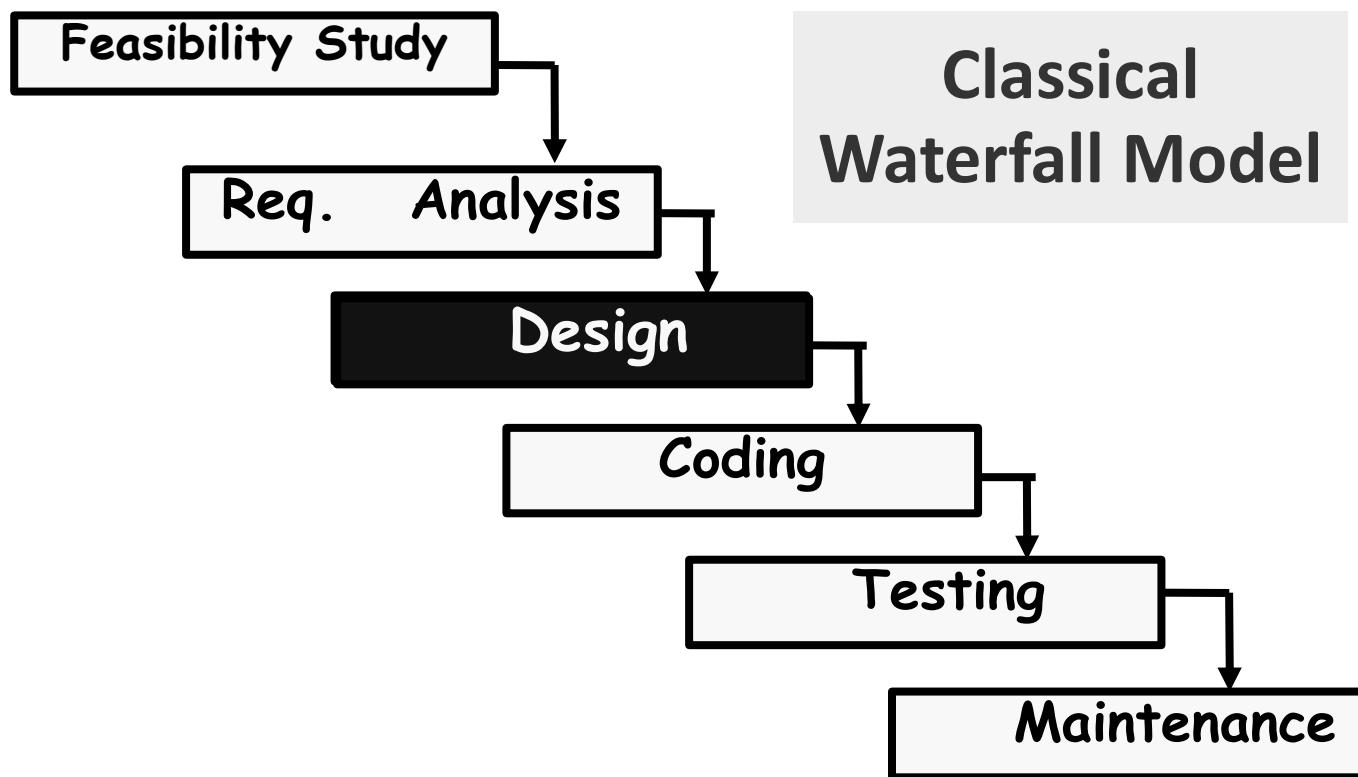
- Gathering relevant data:
 - Usually collected from the end-users through interviews and discussions.
 - Example: for a business accounting software:
 - Interview all the accountants of the organization to find out their requirements.

Requirements Analysis (Cont...)

- The data you initially collect from the users:
 - Usually contain several contradictions and ambiguities.
 - Why?
 - Each user typically has only a partial and incomplete view of the system.**

Requirements Analysis (Cont...)

- Ambiguities and contradictions:
 - must be identified
 - resolved by discussions with the customers.
- Next, requirements are organized:
 - into a Software Requirements Specification (SRS) document.



Design

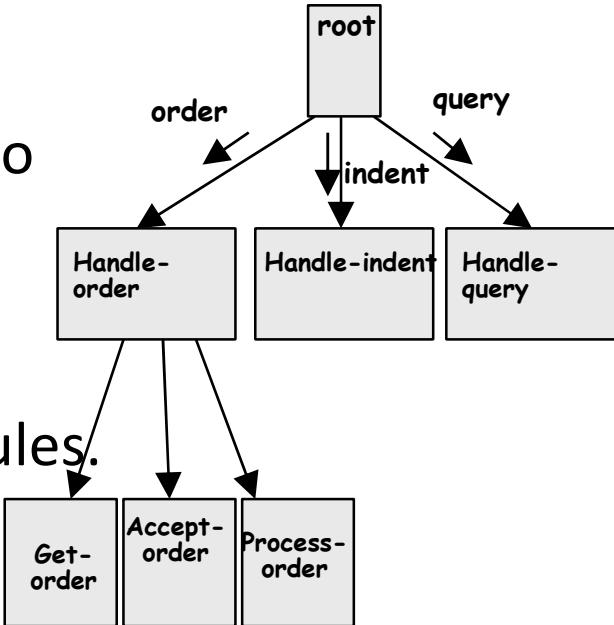
- During design phase requirements specification is transformed into :
 - A form suitable for implementation in some programming language.
- Two commonly used design approaches:
 - Traditional approach,
 - Object oriented approach

Traditional Design Approach

- Consists of two activities:
 - ~~Structured analysis~~ (typically carried out by using DFD)
 - Structured design

Structured Design

- **High-level design:**
 - decompose the system into modules,
 - represent invocation relationships among modules.



- **Detailed design:**
 - different modules designed in greater detail:
 - data structures and algorithms for each module are designed.

- First identify various objects (real world entities) occurring in the problem:
 - Identify the relationships among the objects.
 - For example, the objects in a pay-roll software may be:
 - employees,
 - managers,
 - pay-roll register,
 - Departments, etc.

Object-Oriented Design

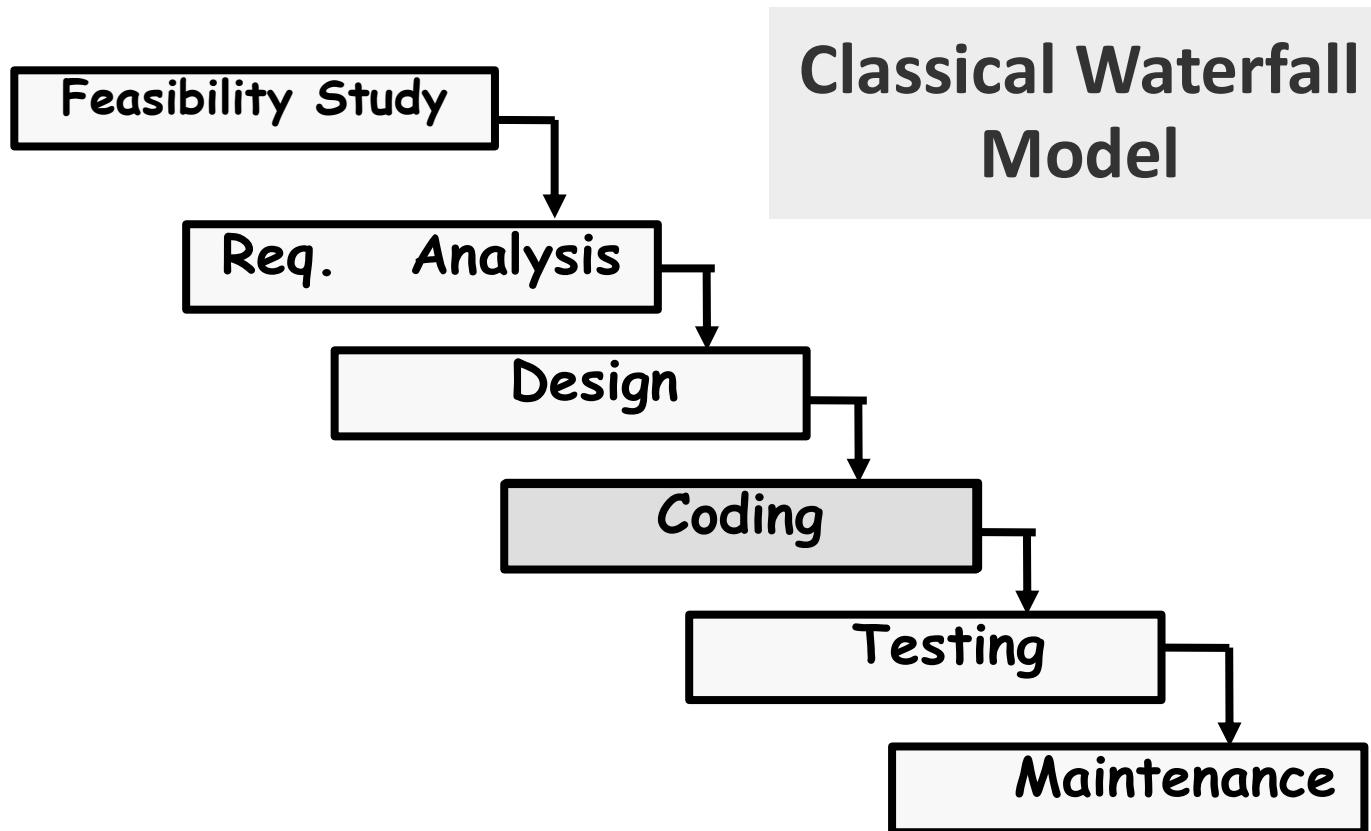
Object Oriented Design (CONT.)

- Object structure:
 - Refined to obtain the detailed design.
- OOD has several advantages:
 - Lower development effort,
 - Lower development time,
 - Better maintainability.

Thank You!!

Life Cycle Models

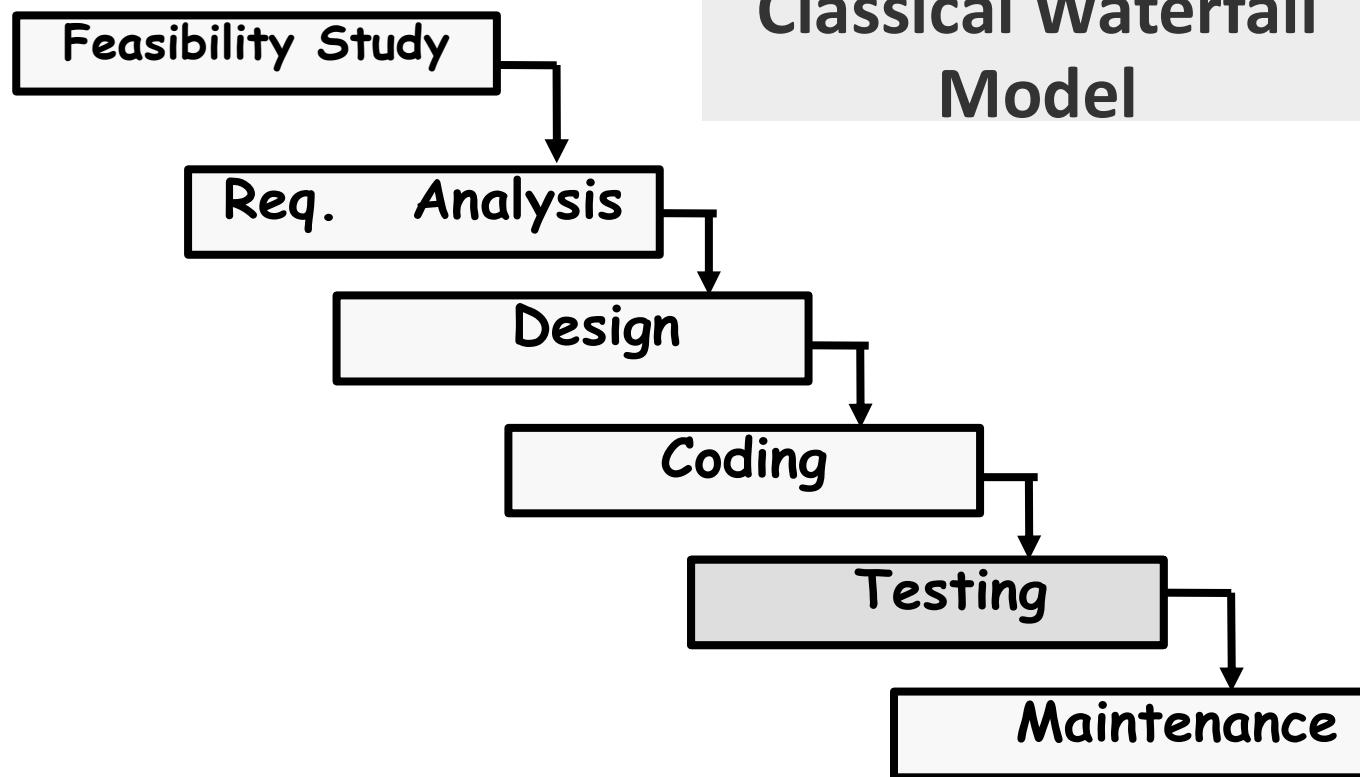
Continued....



Coding and Unit Testing

- During this phase:
 - Each module of the design is coded,
 - Each module is unit tested
 - That is, tested independently as a stand alone unit, and debugged.
 - Each module is documented.

Classical Waterfall Model



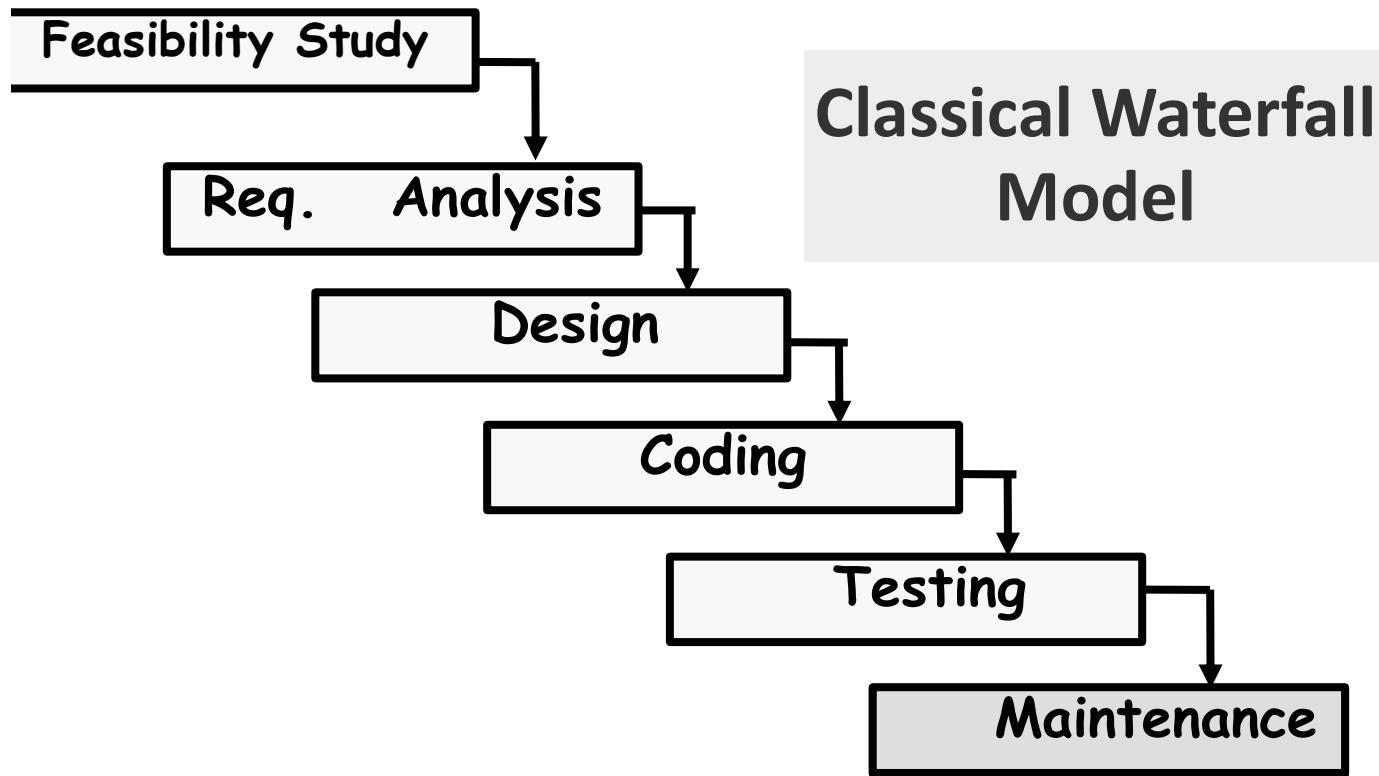
Integration and System Testing

- Different modules are integrated in a planned manner:
 - Modules are usually integrated through a number of steps.
- During each integration step,
 - the partially integrated system is tested.



System Testing

- After all the modules have been successfully integrated and tested:
 - System testing is carried out.
- Goal of system testing:
 - Ensure that the developed system functions according to its requirements as specified in the SRS document.



Maintenance

- Maintenance of any software:
 - Requires much more effort than the effort to develop the product itself.
 - Development effort to maintenance effort is typically 40:60.

Types of Maintenance?

- **Corrective maintenance:**
 - Correct errors which were not discovered during the product development phases.
- **Perfective maintenance:**
 - Improve implementation of the system
 - enhance functionalities of the system.
- **Adaptive maintenance:**
 - Port software to a new environment,
 - e.g. to a new computer or to a new operating system.

Iterative Waterfall Model

- Classical waterfall model is idealistic:
 - Assumes that no defect is introduced during any development activity.
 - In practice:
 - Defects do get introduced in almost every phase of the life cycle.

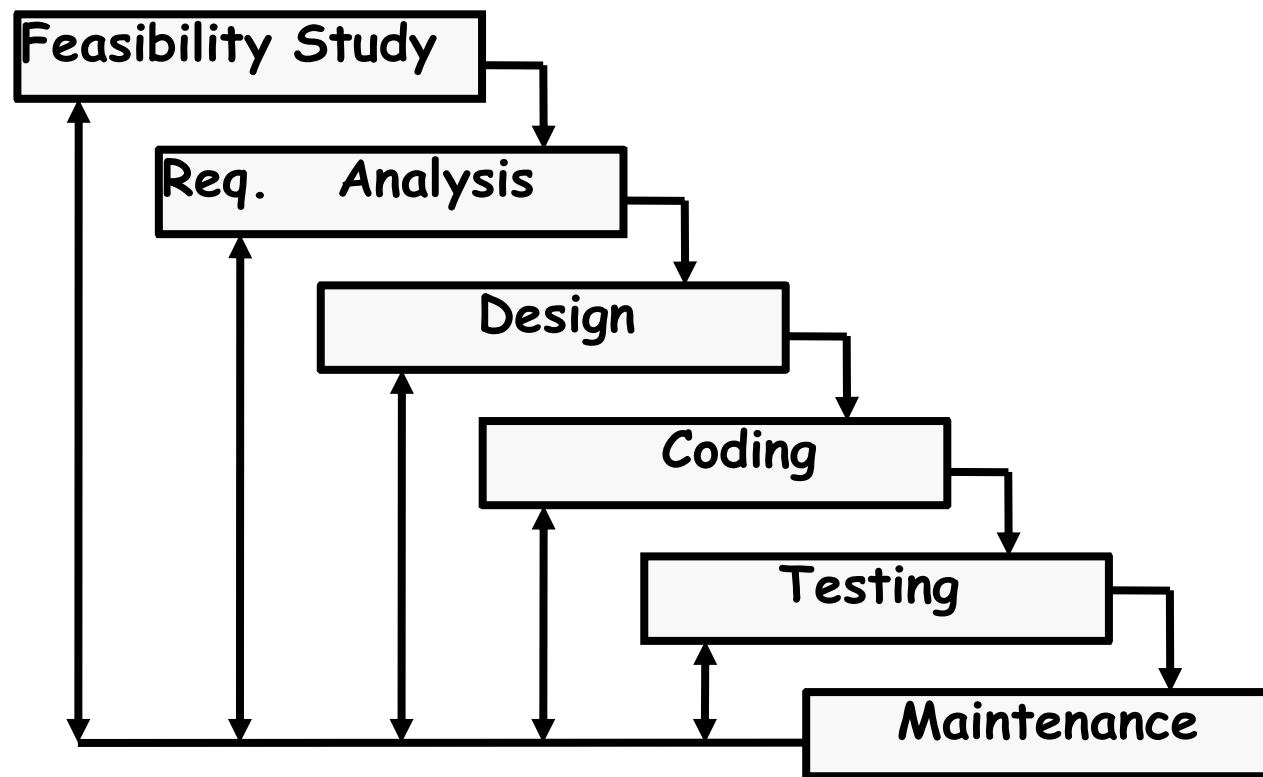
Iterative Waterfall Model (CONT.)

- Defects usually get detected much later in the life cycle:
 - For example, a design defect might go unnoticed till the coding or testing phase.
 - The later the phase in which the defect gets detected, the more expensive is its removal --**
 - why?

Iterative Waterfall Model (CONT.)

- Once a defect is detected:
 - The phase in which it occurred needs to be reworked.
 - Redo some of the work done during that and all subsequent phases.
- Therefore need feedback paths in the classical waterfall model.

Iterative Waterfall Model (CONT.)



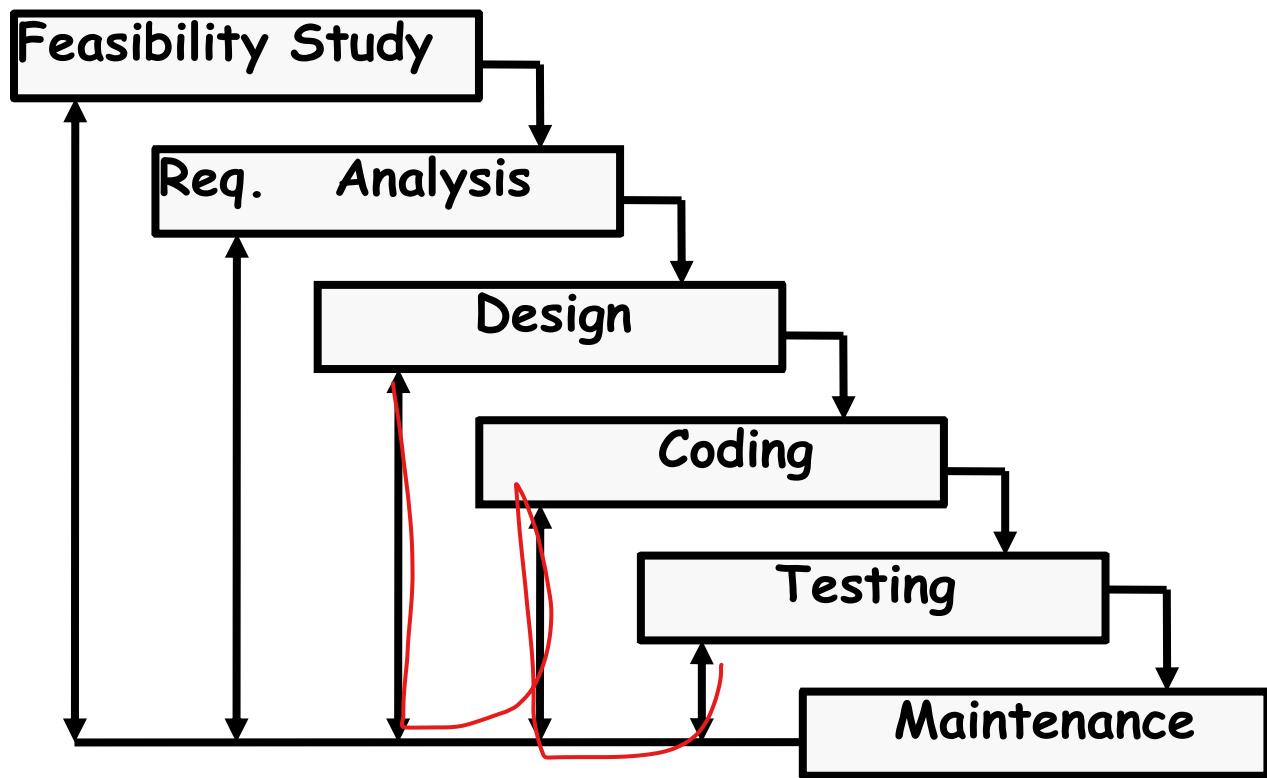
Phase Containment of Errors (Cont...)

- Errors should be detected:
 - In the same phase in which they are introduced.
- For example:
 - If a design problem is detected in the design phase itself,
 - The problem can be taken care of much more easily
 - Than say if it is identified at the end of the integration and system testing phase.

Phase Containment of Errors

- Reason: rework must be carried out not only to the design but also to code and test phases.
- The principle of detecting errors as close to its point of introduction as possible:
 - is known as **phase containment of errors.**
- Iterative waterfall model is by far the most widely used model.
 - Almost every other model is derived from the waterfall model.

Iterative Waterfall Model (CONT.)



Waterfall Strengths

- Easy to understand, easy to use, especially by inexperienced staff
- Milestones are well understood by the team
- Provides requirements stability during development
- Facilitates strong management control (plan, staff, track)

Waterfall Deficiencies

- All requirements must be known upfront –
**in most projects requirement change
occurs after project start**
- Can give a false impression of progress
- Integration is one big bang at the end
- Little opportunity for customer to pre-view
the system.

When to use the Waterfall Model?

- Requirements are well known and stable
- Technology is understood
- Development team have experience with similar projects

Classical Waterfall Model (CONT.)

- Irrespective of the life cycle model actually followed:
 - The documents should reflect a classical waterfall model of development.
 - Facilitates comprehension of the documents.**

Classical Waterfall Model (CONT.)

- Metaphor of mathematical theorem proving:
 - A mathematician presents a proof as a single chain of deductions,
 - Even though the proof might have come from a convoluted set of partial attempts, blind alleys and backtracks.

Requirements Specification and Analysis I

What are Requirements?

- A Requirement is:
 - **A capability or condition required from the system.**
- What is involved in requirements analysis and specification?
 - **Determine what is expected by the client from the system. (Gather and Analyze)**
 - **Document those in a form that is clear to the client as well as to the development team members. (Document)**

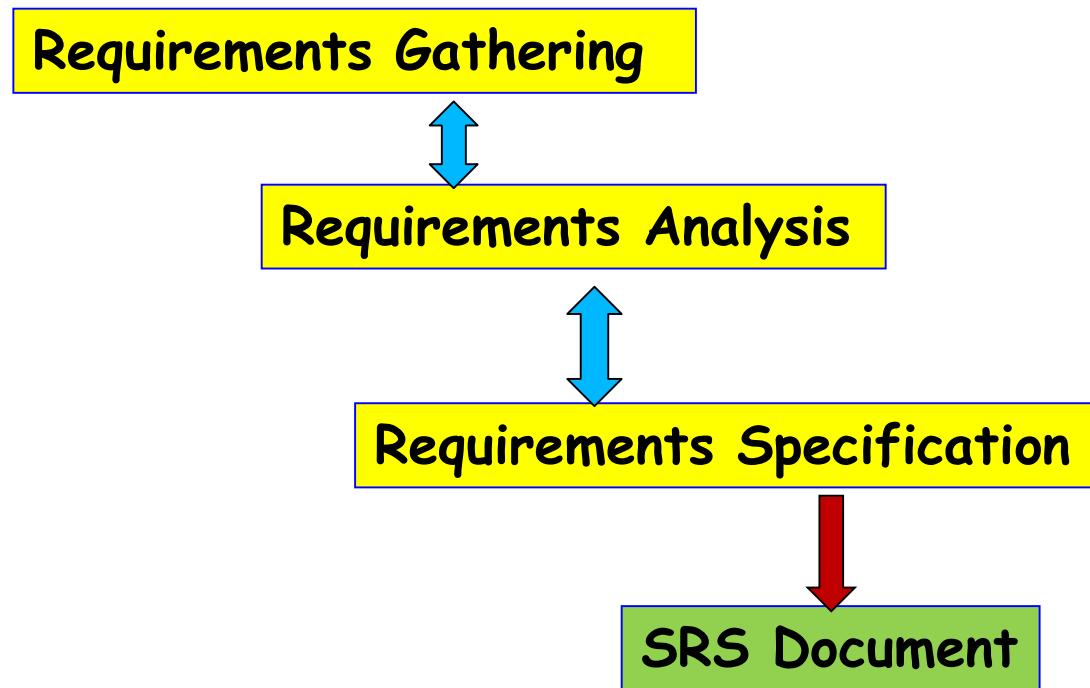
Understanding and specifying requirements

- **For toy problems:** understanding and specifying requirements is rather easy...
- **For industry-standard problems:** Probably the hardest, most problematic and error prone among development tasks...
- The task of requirements specification :
 - **Input:** User needs that are hopefully fully understood by the users.
 - **Output:** Precise statement of what the software will do.

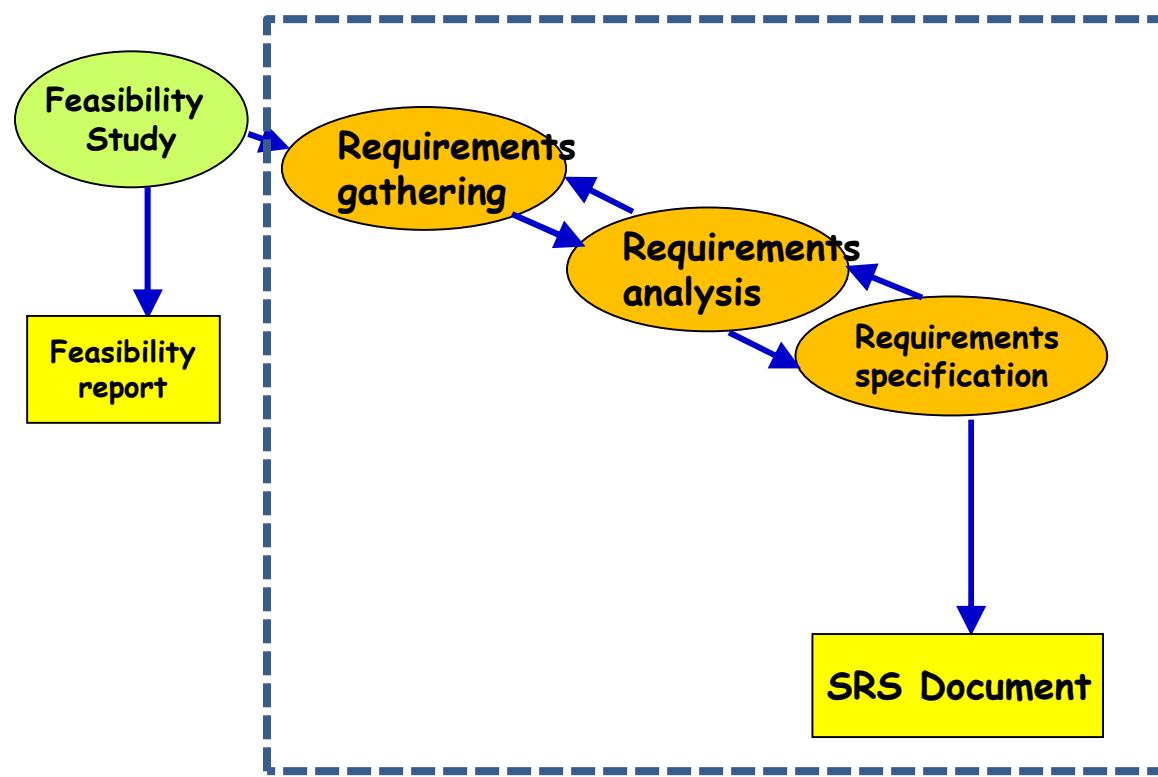
Requirements for Products

- When a company plans to develop a generic product:
 - Who gives the requirements?
- **The sales personnel!**

Activities in Requirements Analysis and Specification



Requirements Engineering Process



Requirements Analysis and Specification

- Requirements Gathering:
 - Fully understand the user requirements.
- Requirements Analysis:
 - Remove inconsistencies, anomalies, etc. from requirements.
- Requirements Specification:
 - Document requirements properly in an SRS document.

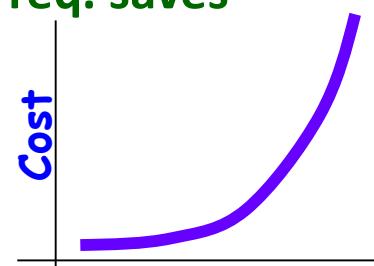
- **Good SRS reduces development cost:**

- Req. errors are expensive to fix later
- Req. changes cost a lot (typically 40% of requirements change later)
- Good SRS can minimize changes and errors
- **Substantial savings --- effort spent during req. saves multiple times that effort**

Need for SRS...

- **An Example:**

- Cost of fixing errors in req. , design , coding , acceptance testing and operation increases exponentially



What are the Uses of an SRS Document?

- Establishes the basis for agreement between the customers and the suppliers
- Forms the starting point for development.
- Provide a basis for estimating costs and schedules.
- Provide a basis for validation and verification.
- Provide a basis for user manual preparation.
- Serves as a basis for later enhancements.

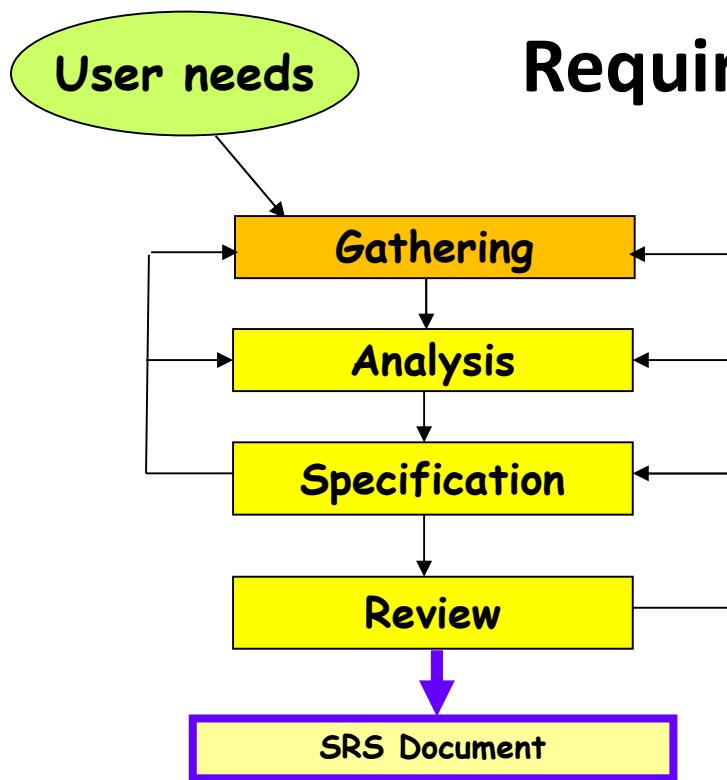
Forms A Basis for User Manual

- The SRS serves as the basis for writing User Manual for the software:
 - **User Manual: Describes the functionality from the perspective of a user --- An important document for users.**
 - Typically also describes how to carry out the required tasks with examples.

- SRS intended for a diverse audience:
 - **Customers and users use it for validation, contract, ...**
 - **Systems (requirements) analysts**
 - **Developers, programmers to implement the system**
 - **Testers use it to check whether requirements have been met**
 - **Project Managers to measure and control the project**
- Different levels of detail and formality is needed for each audience
- Different templates for requirements specifications used by companies:
 - Often variations of **IEEE 830**

**SRS Document:
Stakeholders**

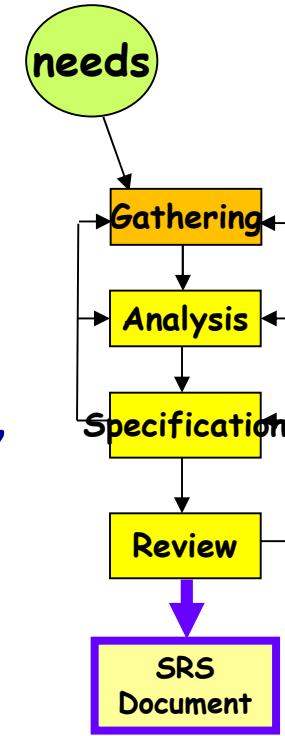
Requirement process..



- Specification and review may lead to further gathering and analysis.

How to Gather Requirements?

- Observe existing (manual) systems,
- Study existing procedures,
- Discuss with customer and end-users,
- Input and Output analysis
- Analyze what needs to be done



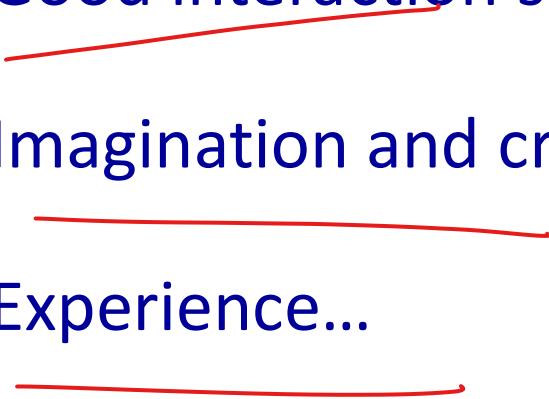
Requirements Gathering Activities

- 1. Study existing documentation
- 2. Interview
- 3. Task analysis
- 4. Scenario analysis
- 5. Form analysis

Requirements Gathering (CONT.)

- In the absence of a working system,
 - Lot of imagination and creativity are required.
- Interacting with the customer to gather relevant data:
 - Requires a lot of experience.

Requirements Gathering (CONT.)

- Some desirable attributes of a good requirements analyst:
 - Good interaction skills,
 - Imagination and creativity,
 - Experience...
- 

Case Study: Automation of Office Work at CSE Dept.

- The academic, inventory, and financial information at the CSE department:
 - At present carried though manual processing by two office clerks, a store keeper, and two attendants.
- Considering the low budget he had at his disposal:
 - The HoD entrusted the work to a team of student volunteers.

Case Study: Automation of Office Work at CSE Dept.

- The team was first briefed by the HoD:
 - Concerning the specific activities to be automated.
- The analysts first discussed with the two office clerks:
 - Regarding their specific responsibilities (tasks) that were to be automated.
Interview
- The analyst also interviewed student and faculty representatives who would also use the software.

Case Study: Automation of Office Work at CSE Dept.

- For each task that a user needs the software to perform, they asked:
 - The steps through which these are to be performed.
 - The various scenarios that might arise for each task.
- Also collected the different types of forms that were being used.

**Task and Scenario
Analysis**

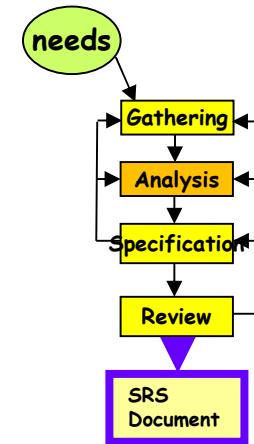
**Form
Analysis**

Case Study: Automation of Office Work at CSE Dept.

- The analysts understood the requirements for the system from various user groups:
Requirements Analysis
 - Identified inconsistencies, ambiguities, incompleteness.
- Resolved the requirements problems through discussions with users:
 - Resolved a few issues which the users were unable to resolve through discussion with the HoD.
- Documented the requirements in the form of an SRS document.
Requirements Specification

Analysis of Gathered Requirements

- Main purpose of req. analysis:
 - Clearly understand user requirements,
 - Detect inconsistencies, ambiguities, and incompleteness.
- Incompleteness and inconsistencies:
 - Resolved through further discussions with the end-users and the customers.



Ambiguity

“When temperature becomes high, start cooler”

Do you notice any problems?

- Above what threshold we consider the temperature to be high?

Inconsistent Requirement

- Some part of the requirement:
 - contradicts some other requirement.
- Example:
 - One customer says turn off heater and open water shower when temperature > 100°C
 - Another customer says turn off heater and turn ON cooler when temperature > 100°C

- Some requirements not included:

- Possibly due to oversight.

- **Example:**

- The analyst has not recorded that when temperature falls below 90°C :

- heater should be turned ON
 - water shower turned OFF.

**Incomplete
Requirement**

Analysis of the Gathered Requirements

- Requirements analysis involves:
 - Obtaining a clear, in-depth understanding of the software to be developed
 - Remove all ambiguities and inconsistencies from the initial customer perception of the problem.

Analysis of the Gathered Requirements (CONT.)

- It is quite difficult to obtain:
 - A clear, in-depth understanding of the problem:
 - Especially if there is no working model of the problem.

Analysis of the Gathered Requirements

(CONT.)

- Experienced analysts take considerable time:
 - Clearly understand the exact requirements the customer has in his mind.



Analysis of the Gathered Requirements

(CONT.)

- Experienced systems analysts know - often as a result of painful experiences ---
 - **“Without a clear understanding of the problem, it is impossible to develop a satisfactory system.”**

Analysis of the Gathered Requirements

- Several things about the project should be clearly understood:
 - What is the problem?
 - What are the possible solutions to the problem?
 - What complexities might arise while solving the problem?

Analysis of the Gathered Requirements

- Some anomalies and inconsistencies can be very subtle:
 - Escape even most experienced eyes.
 - If a formal specification of the system is constructed,
 - Many of the subtle anomalies and inconsistencies get detected.

Analysis of the Gathered Requirements_(CONT.)

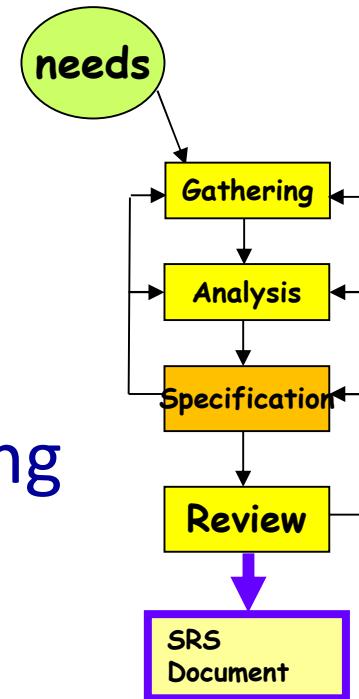
- After collecting all data regarding the system to be developed,
 - Remove all inconsistencies and anomalies from the requirements,
 - Systematically organize requirements into a Software Requirements Specification (SRS) document.

Software Requirements Specification

- Main aim:

- Systematically organize the requirements arrived during requirements analysis.

- Document requirements properly.



SRS Document

- As already pointed out--- useful in various contexts:
 - **Statement of user needs**
 - **Contract document**
 - **Reference document**
 - **Definition for implementation**

SRS Document (CONT.)

- SRS document is known as **black-box specification:**



- The system is considered as a black box whose internal details are not known.
- Only its visible external (i.e. input/output) behavior is documented.

SRS Document (CONT.)

- SRS document concentrates on:
 - **What** needs to be done in terms of input-output behaviour
 - Carefully avoids the solution (“**how to do**”) aspects.

SRS Document (CONT.)

- The requirements at this stage:
 - Written using end-user terminology.
- If necessary:
 - Later a formal requirement specification may be developed from it.

- **It should be concise**
 - and at the same time should not be ambiguous.
- **It should specify what the system must do**
 - and not say how to do it.
- **Easy to change.,**
 - i.e. it should be well-structured.
- **It should be consistent.**
- **It should be complete.**

**Properties of a Good
SRS Document**

Properties of a Good SRS Document

(cont...)

- **It should be traceable**
 - You should be able to trace which part of the specification corresponds to which part of the design, code, etc and vice versa.
- **It should be verifiable**
 - e.g. “system should be user friendly” is not verifiable

SRS should not include...

- **Project development plans**
 - E.g. cost, staffing, schedules, methods, tools, etc
 - Lifetime of SRS is until the software is made obsolete
 - Lifetime of development plans is much shorter
- **Product assurance plans**
 - Configuration Management, Verification & Validation,
test plans, Quality Assurance, etc
 - Different audiences
 - Different lifetimes
- **Designs**
 - Requirements and designs have different audiences
 - Analysis and design are different areas of expertise

Thank You!!

Requirements Specification and Analysis II

SRS Document (CONT.)

- Four important parts:
 - **Functional requirements,**
 - **External Interfaces**
 - **Non-functional requirements,**
 - **Constraints**

Functional Requirements

- Specifies all the functionality that the system should support
 - **Heart of the SRS document:**
 - **Forms the bulk of the Document**
- Outputs for the given inputs and the relationship between them
- Must specify behavior for invalid inputs too!

Functional Requirement Documentation

- **Overview**

- describe purpose of the function and the approaches and techniques employed

- **Inputs and Outputs**

- sources of inputs and destination of outputs
 - quantities, units of measure, ranges of valid inputs and outputs
 - timing

Functional Requirement Documentation

- **Processing**

- validation of input data
- exact sequence of operations
- responses to abnormal situations
- any methods (eg. equations, algorithms)
 - to be used to transform inputs to outputs

Nonfunctional Requirements

- Characteristics of the system which can not be expressed as functions:
 - **Maintainability,**
 - **Portability,**
 - **Usability,**
 - **Security,**
 - **Safety, etc.**

Nonfunctional Requirements

- Reliability issues
- Performance issues:
 - **Example:** How fast can the system produce results?
 - At a rate that does not overload another system to which it supplies data, etc.
 - Response time should be less than 1sec 90% of the time
 - Needs to be measurable (verifiability)

Constraints

- Hardware to be used,
- Operating system
 - or DBMS to be used
- Capabilities of I/O devices
- Standards compliance
- Data representations by the interfaced
system

- User interfaces
- Hardware interfaces
- Software interfaces
- Communications interfaces
with other systems
- File export formats

External Interface Requirements

Goals of Implementation

- Goals describe things that are desirable of the system:
 - But, would not be checked for compliance.
 - For example,
 - Reusability issues
 - Functionalities to be developed in future

IEEE 830-1998 Standard for SRS

- Title
- Table of Contents
- 1. Introduction
 - 1.1 Purpose
 - Describe purpose of the system
 - Describe intended audience
 - 1.2 Scope
 - What the system will and will not do
 - 1.3 Definitions, Acronyms, and Abbreviations
 - Define the vocabulary of the SRS (may also be in appendix)
 - 1.4 References
 - List all referenced documents and their sources SRS (may also be in appendix)
 - 1.5 Overview
 - Describe how the SRS is organized
- 2. Overall Description
- 3. Specific Requirements
- Appendices
- Index

IEEE 830-1998 Standard – Section 2 of

SRS

- Title
- Table of Contents
- 1. Introduction
- 2. Overall Description
 - 2.1 Product Perspective
 - Present the business case and operational concept of the system
 - Describe external interfaces: system, user, hardware, software, communication
 - Describe constraints: memory, operational, site adaptation
 - 2.2 Product Functions
 - 2.3 User Characteristics
 - Summarize the major functional capabilities
 - Describe technical skills of each user class
 - 2.4 Constraints
 - Describe other constraints that will limit developer's options; e.g., regulatory policies; target platform, database, network, development standards requirements
 - 2.5 Assumptions and Dependencies
- 3. Specific Requirements
- 4. Appendices
- 5. Index

IEEE 830-1998 Standard – Section 3 of SRS (1)

- ...
- 1. Introduction
- 2. Overall Description
- 3. Specific Requirements
 - **3.1 External Interfaces**
 - **3.2 Functions**
 - **3.3 Performance Requirements**
 - **3.4 Logical Database Requirements**
 - **3.5 Design Constraints**
 - **3.6 Software System Quality Attributes**
 - **3.7 Object Oriented Models**
- 4. Appendices
- 5. Index

Specify software requirements in sufficient detail so that designers can design the system and testers can verify whether requirements met.

State requirements that are externally perceivable by users, operators, or externally connected systems

Requirements should include, at the least, a description of every input (stimulus) into the system, every output (response) from the system, and all functions performed by the system in response to an input

3.2 Functions

IEEE 830-1998 Standard – Templates

- Section 3 (Specific Requirements) can be organized in several different ways based on
 - **Modes**

 - **User classes**

 - **Concepts (object/class)**

 - **Features**

 - **Stimuli**




Example Section 3 of SRS of

- **SPECIFIC REQUIREMENTS Academic Administration**

3.1 Functional Requirements

Software

3.1.1 Subject Registration

- The subject registration requirements are concerned with functions regarding subject registration which includes students selecting, adding, dropping, and changing a subject.
- **F-001:**
 - The system shall allow a student to register a subject.
- **F-002:**
 - It shall allow a student to drop a course.
- **F-003:**
 - It shall support checking how many students have already registered for a course.

Design Constraints (3.2)

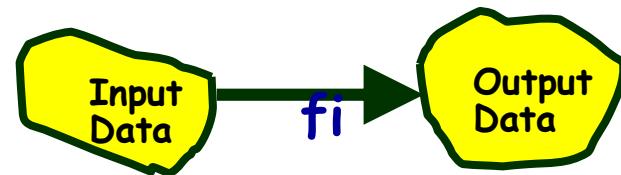
- **3.2 Design Constraints**
- **C-001:**
 - AAS shall provide user interface through standard web browsers.
- **C-002:**
 - AAS shall use an open source RDBMS such as Postgres SQL.
- **C-003:**
 - AAS shall be developed using the JAVA programming language

Non-functional requirements

- **3.3 Non-Functional Requirements**
- **N-001:**
 - AAS shall respond to query in less than 5 seconds.
- **N-002:**
 - AAS shall operate with zero down time.
- **N-003:**
 - AAS shall allow upto 100 users to remotely connect to the system.
- **N-004:**
 - The system will be accompanied by a well-written user manual.

Functional Requirements

- It is desirable to consider every system as:
 - Performing a set of functions $\{f_i\}$.
- Each function f_i considered as:
 - Transforming a set of input data to corresponding output data.

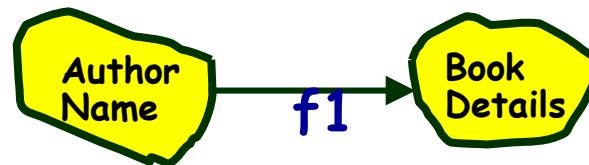


Example: Functional Requirement

- F1: Search Book

- Input:

- an author's name:



- Output:

- details of the author's books and the locations of these books in the library.

Functional Requirements

- Functional requirements describe:
 - A set of high-level requirements
 - Each high-level requirement:
 - takes in some data from the user
 - outputs some data to the user
 - Each high-level requirement:
 - might consist of a set of identifiable sub-functions

Functional Requirements

- For each high-level requirement:
 - A function is described in terms of:
 - Input data set
 - Output data set
 - Processing required to obtain the output data set from the input data set.

Is it a Functional Requirement?

- A high-level function is one:
 - Using which the user can get some useful piece of work done.
- Can the receipt printing work during withdrawal of money from an ATM:
 - Be called a functional requirement?
- A high-level requirement typically involves:
 - Accepting some data from the user,
 - Transforming it to the required response, and then
 - Outputting the system response to the user.

Use Cases

- A use case is a term in UML:
 - Represents a high level functional requirement.
- Use case representation is more well-defined and has agreed documentation:
 - Compared to a high-level functional requirement and its documentation
 - Therefore many organizations document the functional requirements in terms of use cases

Example Functional Requirements

- Req. 1:
 - Once user selects the “search” option,
 - he is asked to enter the key words.
 - The system should output details of all books
 - whose title or author name matches any of the key words entered.
 - Details include: Title, Author Name, Publisher name, Year of Publication, ISBN Number, Catalog Number, Location in the Library.

Example Functional Requirements

- Req. 2:
 - When the “renew” option is selected,
 - The user is asked to enter his membership number and password.
 - After password validation,
 - The list of the books borrowed by him are displayed.
 - The user can renew any of the books:
 - By clicking in the corresponding renew box.

High-Level Function: A Closer View

- A high-level function:
 - Usually involves a series of interactions between the system and one or more users.
- Even for the same high-level function,
 - There can be different interaction sequences (or scenarios)
 - Due to users selecting different options or entering different data items.

Examples of Bad SRS Documents

Unstructured Specifications:

– **Narrative essay --- one of the worst types of specification document:**

- Difficult to change,
- Difficult to be precise,
- Difficult to be unambiguous,
- Scope for contradictions, etc.

Examples of Bad SRS Documents

- **Noise:**

- Presence of text containing information irrelevant to the problem.

- **Silence:**

- Aspects important to proper solution of the problem are omitted.

Examples of Bad SRS

- **Overspecification:** Documents
 - Addressing “how to” aspects
 - For example, “Library member names should be stored in a sorted descending order”
 - Overspecification restricts the solution space for the designer.
- **Contradictions:**
 - Contradictions might arise
 - if the same thing described at several places in different ways.

Examples of Bad SRS Documents

- **Ambiguity:**

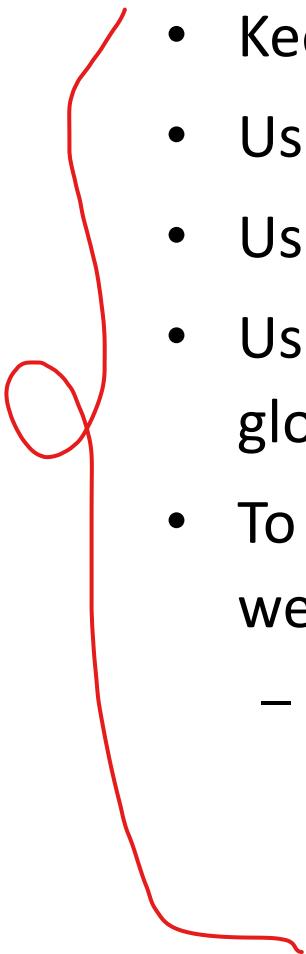
- Literary expressions
- Unquantifiable aspects, e.g. “good user interface”

- **Forward References:**

- References to aspects of problem
 - defined only later on in the text.

- **Wishful Thinking:**

- Descriptions of aspects
 - for which realistic solutions will be hard to find.



Suggestions for Writing Good Quality Requirements

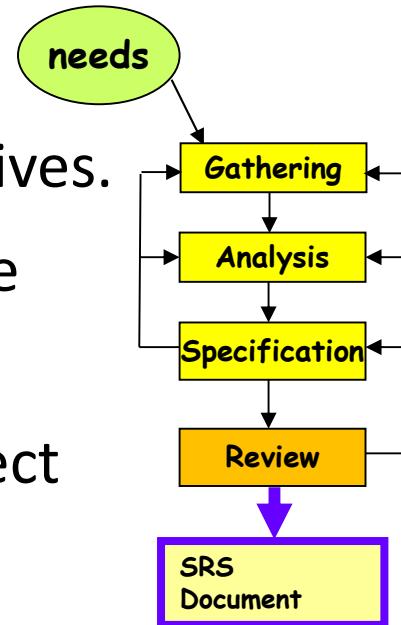
- Keep sentences and paragraphs short.
- Use active voice.
- Use proper grammar, spelling, and punctuation.
- Use terms consistently and define them in a glossary.
- To see if a requirement statement is sufficiently well defined,
 - Read it from the developer's perspective

Suggestions for Writing Good Quality Requirements

- Split a requirement into multiple sub-requirements:
 - Because each will require separate test cases and because each should be separately traceable.
 - If several requirements are strung together in a paragraph, it is easy to overlook one during construction or testing.

SRS Review

- **Review** done by the Developers along with the user representatives.
- To verify that SRS confirms to the actual user requirements
- To detect defects early and correct them.
- Review typically done using standard inspection process:
 - Checklists.



A Sample SRS Checklist

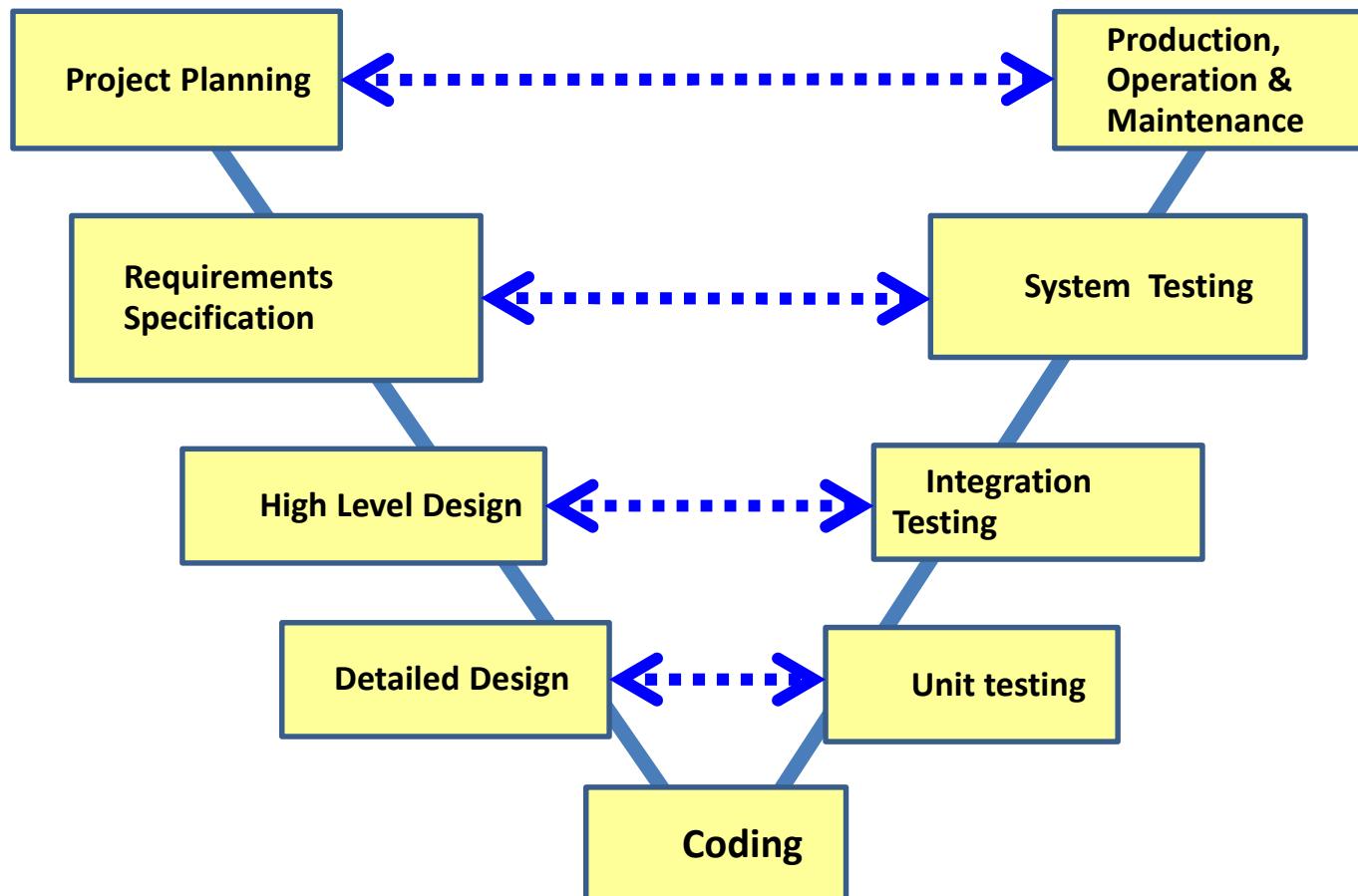
- Have response times been specified for functions ?
- Have all the HW, external SW and data interfaces been defined ?
- Is each requirement testable ?
- Is the initial state of the system defined ?
- Are the responses to exceptional conditions specified ?

Thank You!!

V Model

V Model

- It is a variant of the Waterfall
 - emphasizes verification and validation
 - V&V activities are spread over the entire life cycle.
- In every phase of development:
 - Testing activities are planned in parallel with development.



V Model Steps

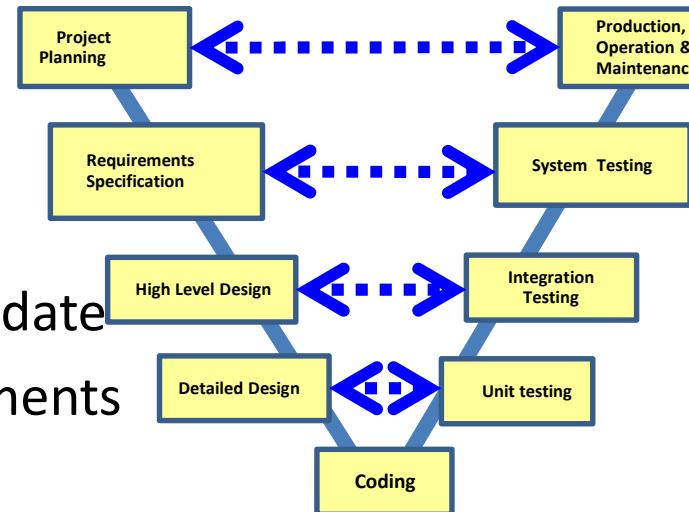
- Planning
- Requirements Analysis and Specification • System test design
- High-level Design • Integration Test design
- Detailed Design • Unit test design

V Model: Strengths

- Starting from early stages of software development:
 - Emphasizes planning for verification and validation of the software
- Each deliverable is made testable
- Easy to use

V Model Weaknesses

- Does not support overlapping of phases
- Does not handle iterations or phases
- Does not easily accommodate later changes to requirements
- Does not provide support for effective risk handling



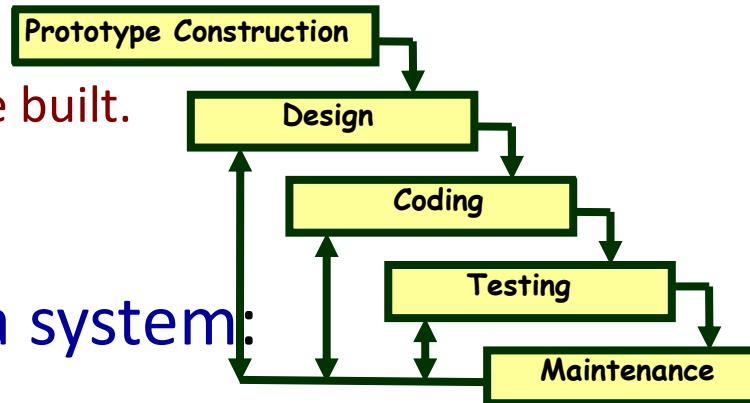
When to use V Model

- Natural choice for systems requiring high reliability:
 - Embedded control applications, safety-critical software
- All requirements are known up-front
- Solution and technology are known

Prototyping Model

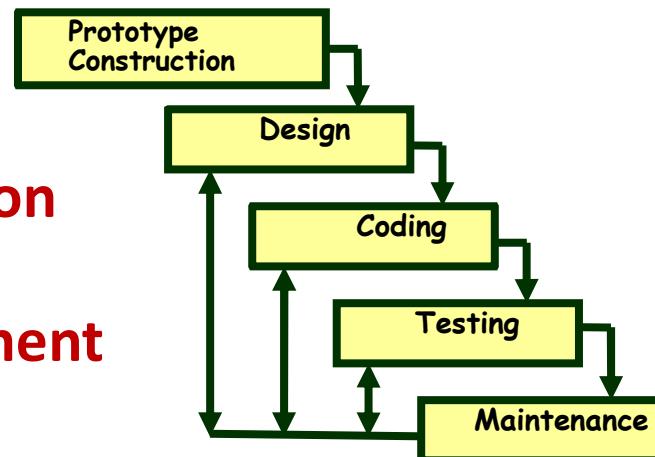
Prototyping Model

- A derivative of waterfall model.
- Before starting actual development,
 - A working prototype of the system should first be built.
- A prototype is a toy implementation of a system:
 - Limited functional capabilities,
 - Low reliability,
 - Inefficient performance.



Reasons for prototyping

- **Learning by doing:** useful where requirements are only partially known
- **Improved communication**
- **Improved user involvement**
- **Reduced need for documentation**
- **Reduced maintenance costs**



Reasons for Developing a Prototype

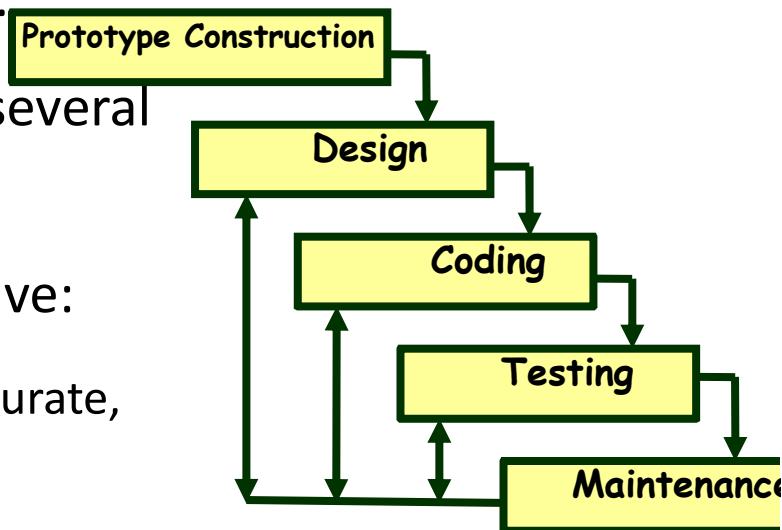
- **Illustrate to the customer:**
 - input data formats, messages, reports, or interactive dialogs.
- **Examine technical issues associated with product development:**
 - Often major design decisions depend on issues like:
 - Response time of a hardware controller,
 - Efficiency of a sorting algorithm, etc.

Prototyping Model (CONT.)

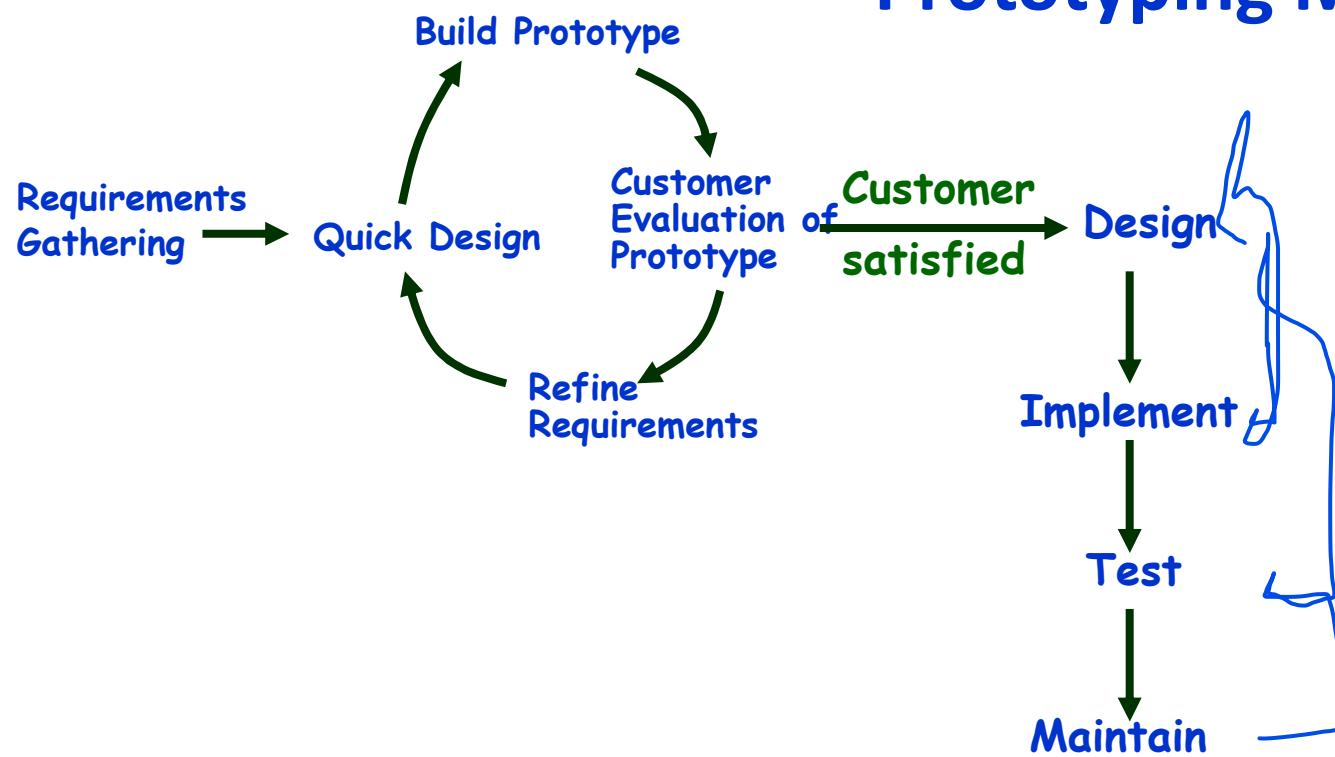
- Another reason for developing a prototype:
 - It is impossible to “get it right” the first time,**
 - We must plan to throw away the first version:
 - If we want to develop a good software.

Prototyping Model

- Start with approximate requirements.
- Carry out a quick design.
- Prototype is built using several short-cuts:
 - Short-cuts might involve:
 - Using inefficient, inaccurate, dummy functions.
 - A table look-up rather than performing the actual computations.



Prototyping Model



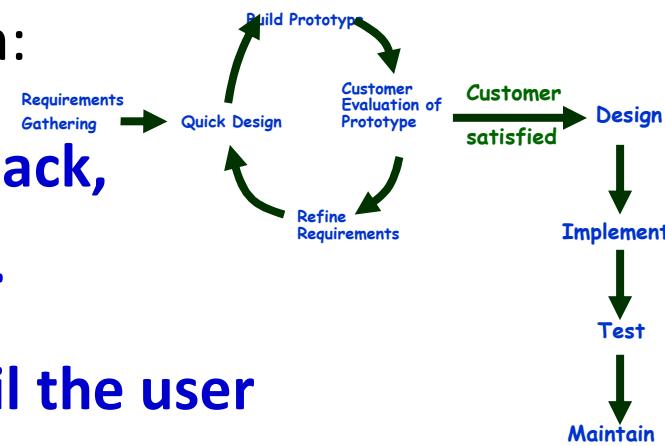
Prototyping Model (CONT.)

- The developed prototype is submitted to the customer for his evaluation:

**—Based on the user feedback,
the prototype is refined.**

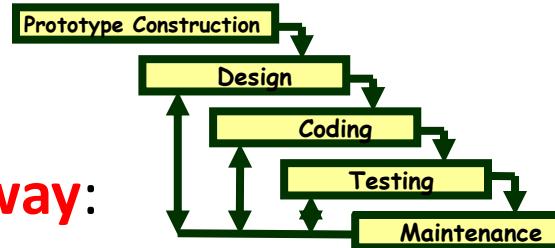
**—This cycle continues until the user
approves the prototype.**

- The actual system is developed using the waterfall model.



Prototyping Model (CONT.)

- Requirements analysis and specification phase becomes redundant:
 - Final working prototype (incorporating all user feedbacks) serves as an animated requirements specification.
- Design and code for the prototype is usually thrown away:
 - However, experience gathered from developing the prototype helps a great deal while developing the actual software.



Prototyping Model (CONT.)

- Even though construction of a working prototype model involves additional cost --- **overall development cost usually lower for:**
 - Systems with unclear user requirements,
 - Systems with unresolved technical issues.
- Many user requirements get properly defined and technical issues get resolved:
 - These would have appeared later as change requests and resulted in incurring massive redesign costs.

Prototyping: advantages

- The resulting software is usually more usable
- User needs are better accommodated
- The design is of higher quality
- The resulting software is easier to maintain
- Overall, the development incurs less cost

Prototyping: disadvantages

- For some projects, it is expensive
- Susceptible to over-engineering:
 - Designers start to incorporate sophistications that they could not incorporate in the prototype.

Major difficulties of Waterfall-Based Models

1. Difficulty in accommodating change requests during development.
 - **40% of the requirements change during development**
2. High cost incurred in developing custom applications.
3. **“Heavy weight processes.”**

Major difficulties of Waterfall-Based Life Cycle Models

- Requirements for the system are determined at the start:
 - Are assumed to be fixed from that point on.
 - Long term planning is made based on this.

“... the assumption that one can specify a
satisfactory system in advance, get bids for
its construction, have it built, and install it.
...this assumption is fundamentally wrong
and many software acquisition problems
spring from this...”

Frederick Brooks

Thank You!!

Life Cycle Models: Incremental, Evolutionary, and RAD Models

- “The basic idea... take advantage of what was being learned during the development of earlier, incremental, deliverable versions of the system. Learning comes from both the development and use of the system... Start with a simple implementation of a subset of the software requirements and iteratively enhance the evolving sequence of versions. At each version design modifications are made along with adding new functional capabilities.” **Victor Basili**

Incremental and Iterative Development (IID)

- **Key characteristics**

- Builds system incrementally
- Consists of a planned number of iterations
- Each iteration produces a working program

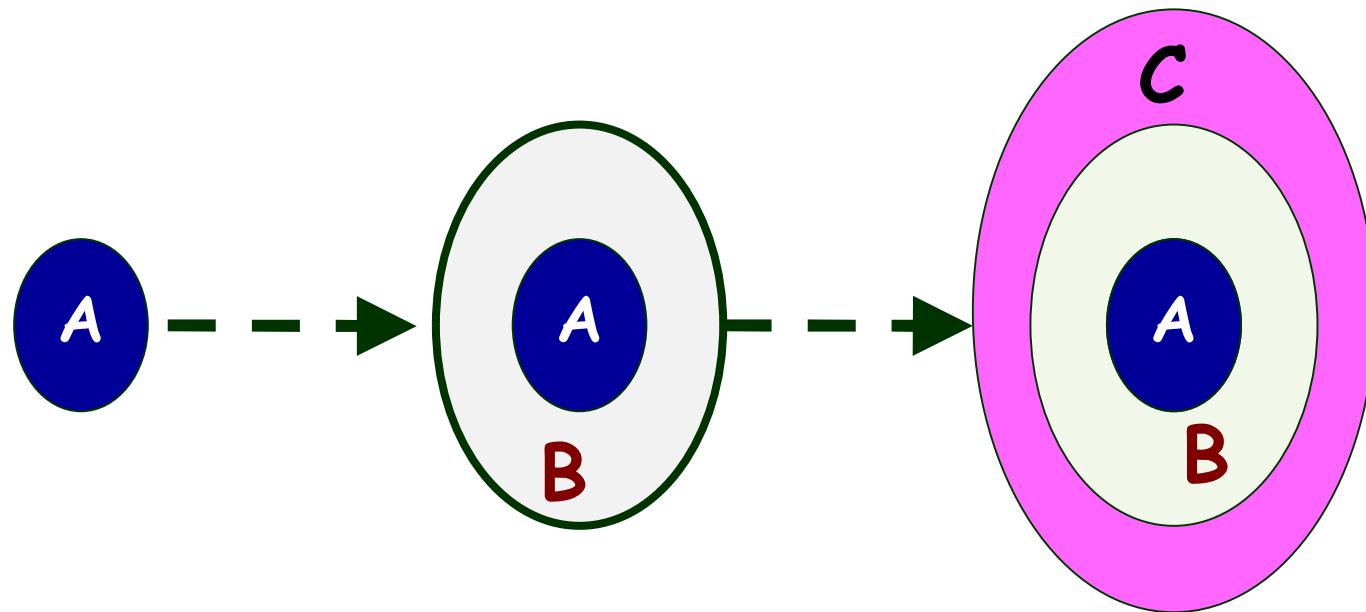
- **Benefits**

- Facilitates and manages changes

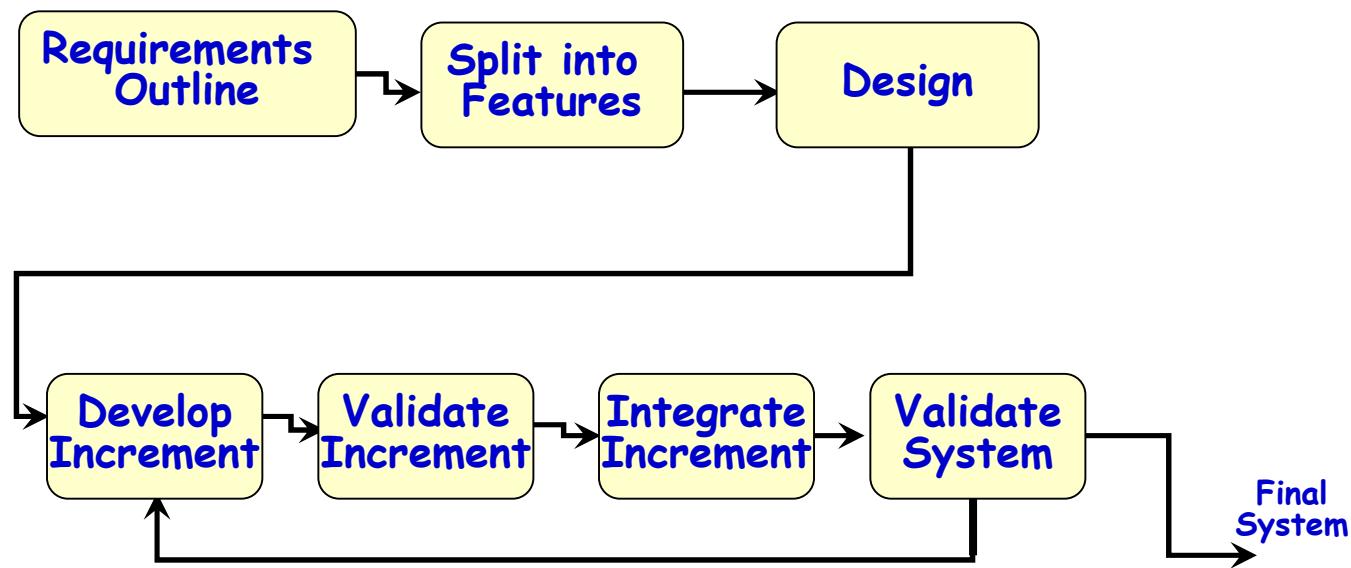
- **Foundation of agile techniques and the basis for**

- Rational Unified Process (RUP)
- Extreme Programming (XP)

Customer's Perspective



Incremental Model



Incremental Model: Requirements

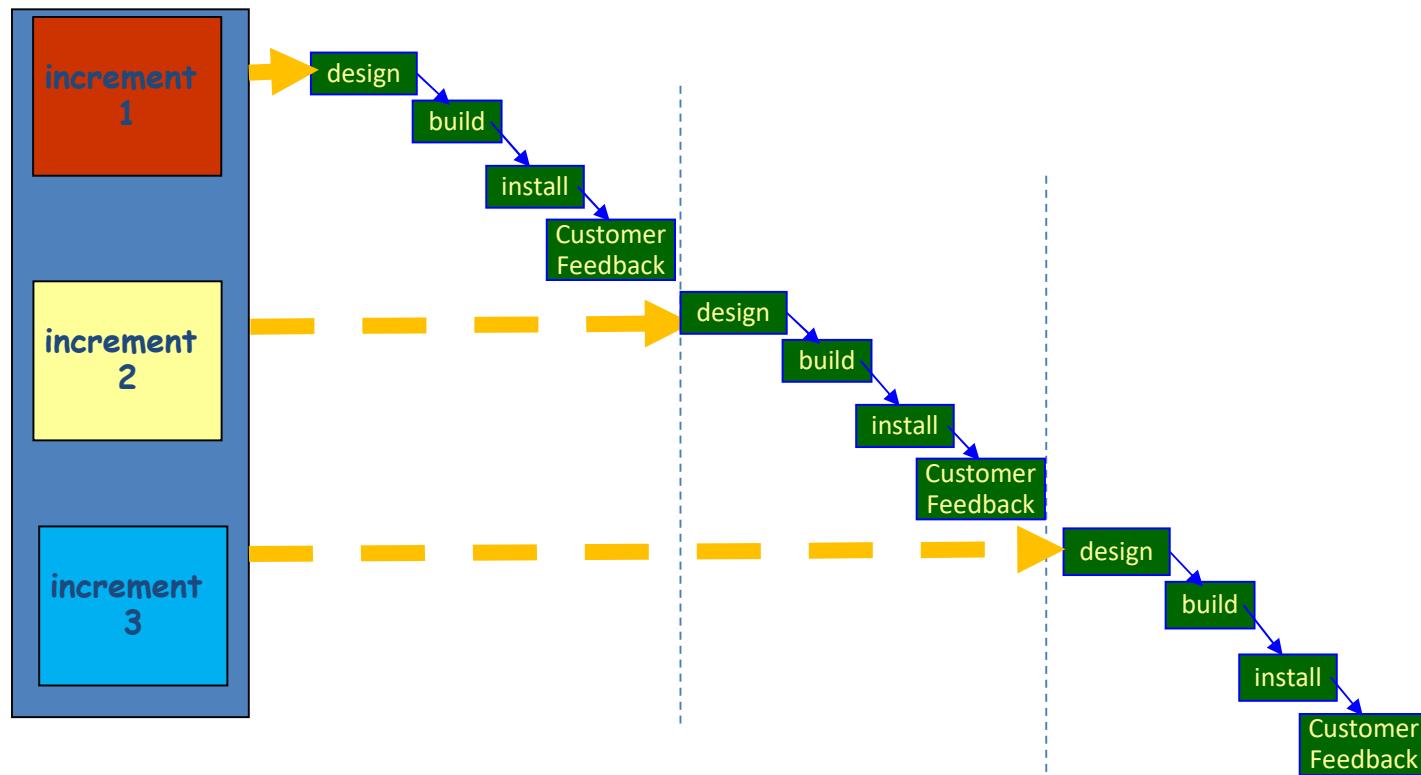
Split into
Features



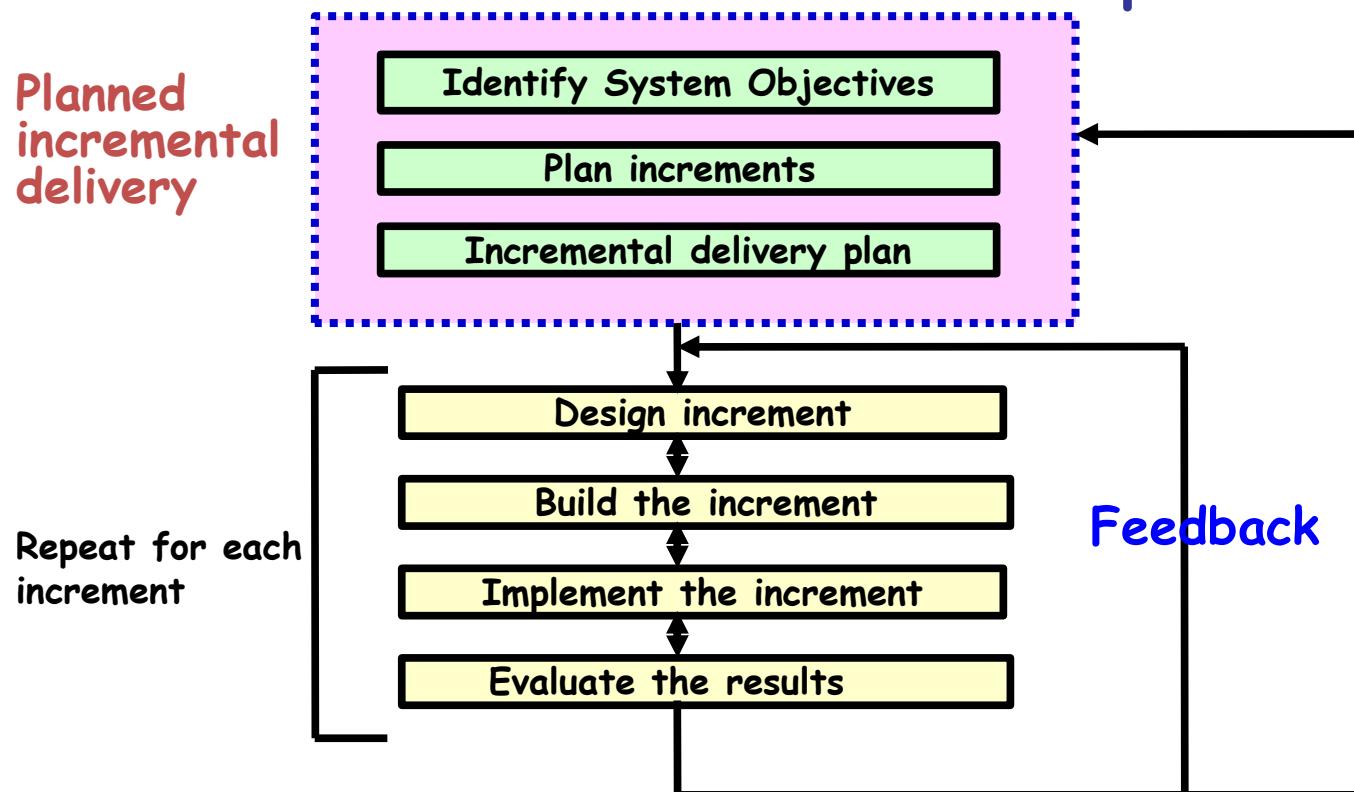
Incremental Model

- Waterfall: single release
- Iterative: many releases (increments)
 - First increment: core functionality
 - Successive increments: add/fix functionality
 - Final increment: the complete product
- Each iteration: a short mini-project with a separate lifecycle
 - e.g., waterfall

Incremental delivery



Incremental process



Which step first?

- Some steps will be pre-requisite because of physical dependencies
- Others may be in any order
- Value to cost ratios may be used
 - V/C where
 - V is a score 1-10 representing value to customer
 - C is a score 0-10 representing cost to developers

V/C ratios: an example

step	value	cost	ratio	
profit reports	9	2	4.5	2nd
online database	1	9	0.11	5th
ad hoc enquiry	5	5	1	4th
purchasing plans	9	4	2.25	3rd
profit-based pay for managers	9	1	9	1st

Evolutionary Model with Iterations

An Evolutionary and Iterative Development Process...

- Recognizes the reality of changing requirements
 - Capers Jones's research on 8000 projects: **40% of final requirements arrived after development had already begun**
- Promotes early risk mitigation:
 - Breaks down the system into mini-projects and focuses on the riskier issues first.
 - “**plan a little, design a little, and code a little”**
- Encourages all development participants to be involved earlier on,:
 - End users, Testers, integrators, and technical writers

Evolutionary Model with Iteration

- “A complex system will be most successful if implemented in small steps... “retreat” to a previous successful step on failure... opportunity to receive some feedback from the real world before throwing in all resources... and you can correct possible errors...” **Tom Glib in Software Metrics**

Evolutionary model with iteration

- Evolutionary iterative development implies that the requirements, plan, estimates, and solution evolve or are refined over the course of the iterations, rather than fully defined and “frozen” in a major up-front specification effort before the development iterations begin. Evolutionary methods are consistent with the pattern of unpredictable discovery and change in new product development.” **Craig Larman**

Evolutionary Model

- First develop the core modules of the software.
- The initial skeletal software is refined into increasing levels of capability:
(Iterations)
 - By adding new functionalities in successive versions.

Activities in an Iteration

- Software developed over several “mini waterfalls”.
- The result of a single iteration:
 - Ends with delivery of some tangible code
 - An incremental improvement to the software --- leads to evolutionary development

Evolutionary Model with Iteration

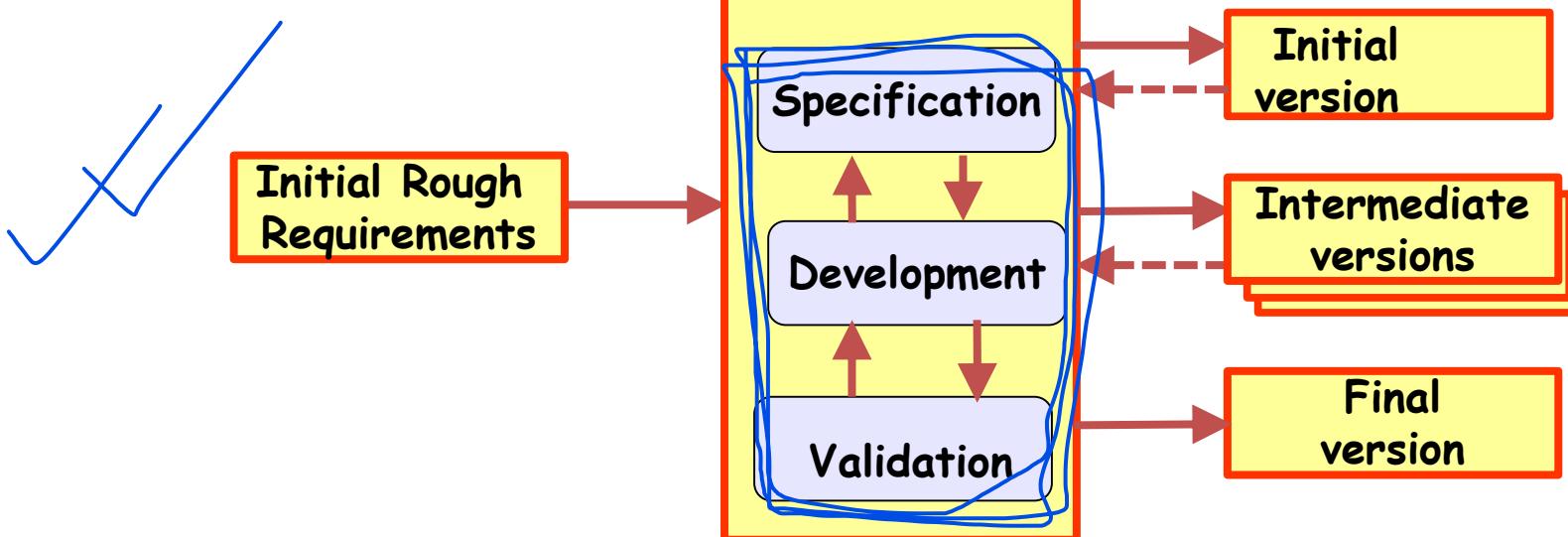
- Outcome of each iteration: tested, integrated, executable system
- Iteration length is short and fixed
 - **Usually between 2 and 6 weeks**
 - **Development takes many iterations (for example: 10-15)**
- Does not “freeze” requirements and then conservatively design :
 - **Opportunity exists to modify requirements as well as the design...**

Evolutionary Model (CONT.)

- Successive versions:
 - Functioning systems capable of performing some useful work.
 - A new release may include new functionality:
 - Also existing functionality in the current release might have been enhanced.

Evolutionary Model

- Evolves an initial implementation with user feedback:
 - **Multiple versions until the final version.**



Advantages of Evolutionary Model

- Users get a chance to experiment with a partially developed system:
 - Much before the full working version is released,
- **Helps finding exact user requirements:**
 - Software more likely to meet exact user requirements.
- **Core modules get tested thoroughly:**
 - Reduces chances of errors in final delivered software.

Advantages of evolutionary model

- Better management of complexity by developing one increment at a time.
- Better management of changing requirements.
- Can get customer feedback and incorporate them much more efficiently:
 - As compared when customer feedbacks come only after the development work is complete.

Advantages of Evolutionary Model with Iteration

- Training can start on an earlier release
 - customer feedback taken into account
- Frequent releases allow developers to fix unanticipated problems quicker.

Evolutionary Model: Problems

- **The process is intangible:**
 - No regular, well-defined deliverables.
- **The process is unpredictable:**
 - Hard to manage, e.g., scheduling, workforce allocation, etc.
- **Systems are rather poorly structured:**
 - Continual, unpredictable changes tend to degrade the software structure.
- **Systems may not even converge to a final version.**

RAD Model

Rapid Application Development (RAD) Model

- Sometimes referred to as the **rapid prototyping model.**
- Major aims:
 - Decrease the time taken and the cost incurred to develop software systems.
 - Facilitate accommodating change requests as early as possible:
 - Before large investments have been made in development and testing.

Important Underlying Principle

- A way to reduce development time and cost, and yet have flexibility to incorporate changes:

- **Make only short term plans and make heavy reuse of existing code.**

Methodology

- Plans are made for one increment at a time.
 - The time planned for each iteration is called a **time box**.
- Each iteration (increment):
 - Enhances the implemented functionality of the application a little.

Methodology

- During each iteration,
 - A quick-and-dirty prototype-style software for some selected functionality is developed.
 - The customer evaluates the prototype and gives his feedback.
 - The prototype is refined based on the customer feedback.

How Does RAD Facilitate Faster Development?

- RAD achieves fast creation of working prototypes.
 - Through use of specialized tools.
- These specialized tools usually support the following features:
 - Visual style of development.
 - Use of reusable components.
 - Use of standard APIs (Application Program Interfaces).

For which Applications is RAD Suitable?

- Customized product developed for one or two customers only
- Performance and reliability are not critical.
- The system can be split into several independent modules.

For Which Applications RAD is Unsuitable?

- Few plug-in components are available
- High performance or reliability required
- No precedence for similar products exists
- The system cannot be modularized.

Prototyping versus RAD

- In prototyping model:
 - The developed prototype is primarily used to gain insights into the solution
 - Choose between alternatives
 - Elicit customer feedback.
- The developed prototype:
 - Usually thrown away.

Prototyping versus RAD

- In contrast:
 - In RAD the developed prototype evolves into deliverable software.
 - RAD leads to faster development compared to traditional models:
 - However, the quality and reliability would possibly be poorer.

RAD versus Iterative Waterfall Model

- In the iterative waterfall model,
 - All product functionalities are developed together.
- In the RAD model on the other hand,
 - Product functionalities are developed incrementally through heavy code and design reuse.
 - Customer feedback is obtained on the developed prototype after each iteration:
 - Based on this the prototype is refined.

RAD versus Iterative Waterfall Model

- The iterative waterfall model:
 - Does not facilitate accommodating requirement change requests.
- Iterative waterfall model does have some important advantages:
 - Use of the iterative waterfall model leads to production of good documentation.
 - Also, the developed software usually has better quality and reliability than that developed using RAD.

- Incremental development:
 - Occurs in both evolutionary and RAD models.
- However, in RAD:
 - Each increment is a quick and dirty prototype,
 - Whereas in the evolutionary model each increment is systematically developed using the iterative waterfall model.
- Also, RAD develops software in shorter increments:
 - The incremental functionalities are fairly large in the evolutionary model.

**RAD versus
Evolutionary
Model**

Thank You!!

Life Cycle Models

cont...

Agile Models

What is Agile Software Development?

- Agile: Easily moved, light, nimble, active software processes
- How agility achieved?
 - Fitting the process to the project
 - Avoidance of things that waste time

Agile Model

- To overcome the shortcomings of the waterfall model of development.
 - Proposed in mid-1990s
- The agile model was primarily designed:
 - To help projects to adapt to change requests
- In the agile model:
 - The requirements are decomposed into many small incremental parts that can be developed over one to four weeks each.

Ideology: Agile Manifesto

- Individuals and interactions *over*
 - process and tools <http://www.agilemanifesto.org>
- Working Software *over*
 - comprehensive documentation
- Customer collaboration *over*
 - contract negotiation
- Responding to change *over*
 - following a plan

Agile Methodologies

- XP
- Scrum
- Unified process
- Crystal
- DSDM
- Lean

Agile Model: Principal Techniques

- **User stories:**

- Simpler than use cases.

- **Metaphors:**

- Based on user stories, developers propose a common vision of what is required.

- **Spike:**

- Simple program to explore potential solutions.

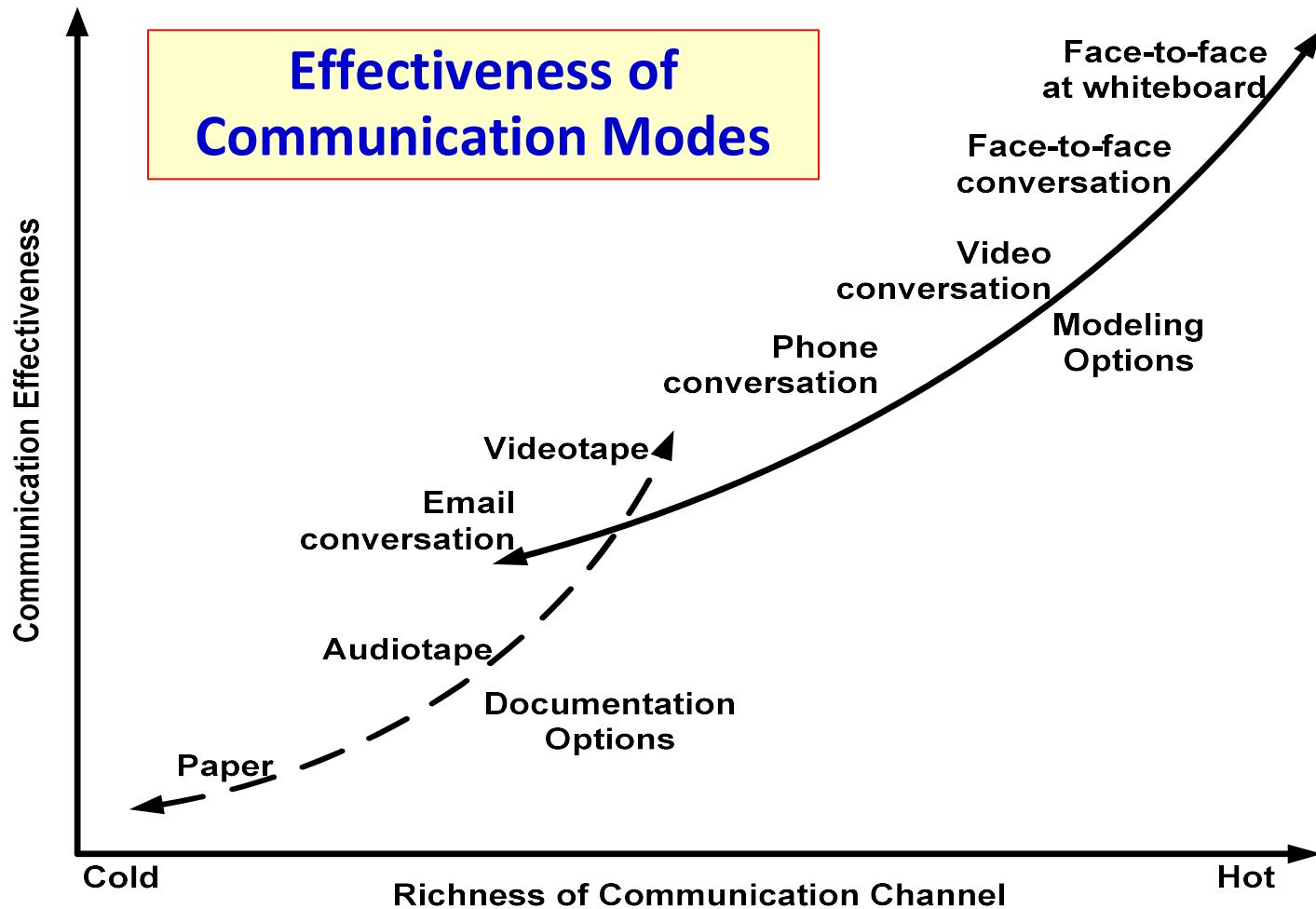
- **Refactor:**

- Restructure code without affecting behavior, improve efficiency, structure, etc.

- At a time, only one increment is planned, developed, deployed at the customer site.
 - No long-term plans are made.
- An iteration may not add significant functionality,
Agile Model: Nitty Gritty
 - But still a new release is invariably made at the end of each iteration
 - Delivered to the customer for regular use.

Methodology

- Face-to-face communication favoured over written documents.
- To facilitate face-to-face communication,
 - Development team to share a single office space.
 - Team size is deliberately kept small (5-9 people)
 - This makes the agile model most suited to the development of small projects.



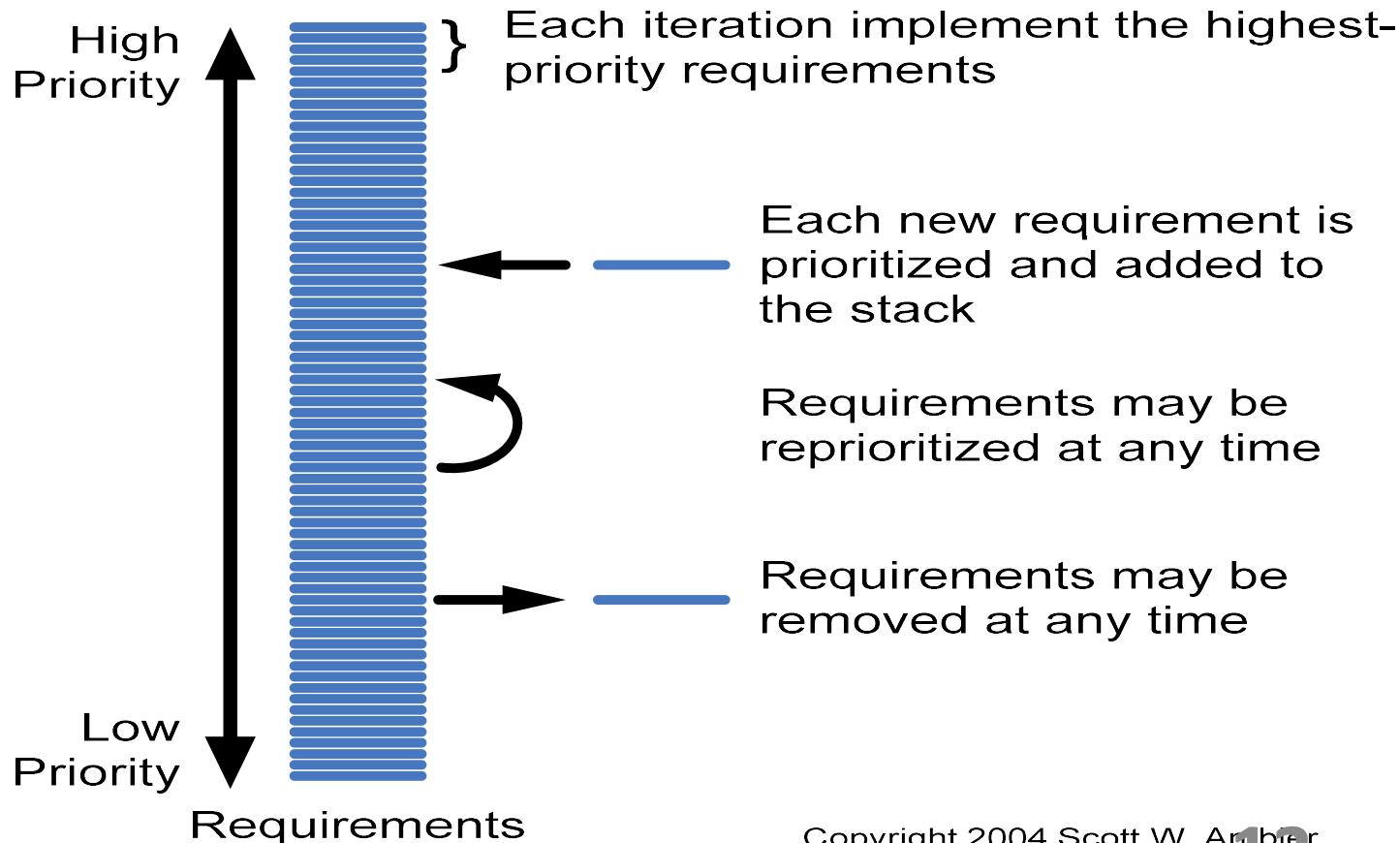
Copyright 2002-2005 Scott W. Ambler
Original Diagram Copyright 2002 Alistair Cockburn

Agile Model: Principles

- The primary measure of progress:
 - Incremental release of working software
- Important principles behind agile model:
 - Frequent delivery of versions --- once every few weeks.
 - Requirements change requests are easily accommodated.
 - Close cooperation between customers and developers.
 - Face-to-face communication among team members.

- Travel light:
 - You need far less documentation than you think.
- Agile documents:
 - Are concise
 - Describe information that is less likely to change
 - Describe “good things to know”
 - Are sufficiently accurate, consistent, and detailed
- Valid reasons to document:
 - Project stakeholders require it
 - To define a contract model
 - To support communication with an external group
 - To think something through

Agile Software Requirements Management



Adoption Detractors

- Sketchy definitions, make it possible to have
 - Inconsistent and diverse definitions
- High quality people skills required
- Short iterations inhibit long-term perspective
- Higher risks due to feature creep:
 - Harder to manage feature creep and customer expectations

Agile Model Shortcomings

- Derives agility through developing tacit knowledge within the team, rather than any formal document:
 - Can be misinterpreted...
 - External review difficult to get...
 - When project is complete, and team disperses, maintenance becomes difficult...

- Steps of **Agile Model versus Waterfall Model**

Waterfall model are a planned sequence:

- Requirements-capture, analysis, design, coding, and testing .
- Progress is measured in terms of delivered artefacts:
 - Requirement specifications, design documents, test plans, code reviews, etc.
- In contrast agile model sequences:
 - Delivery of working versions of a product in several increments.

Agile Model versus Iterative Waterfall Model

- As regards to similarity:
 - We can say that Agile teams use the waterfall model on a small scale.

Agile versus RAD Model

- Agile model does not recommend developing prototypes:
 - Systematic development of each incremental feature is emphasized.
- In contrast:
 - RAD is based on designing quick-and-dirty prototypes, which are then refined into production quality code.

NO

Agile versus exploratory programming

- Similarity:
 - Frequent re-evaluation of plans,
 - Emphasis on face-to-face communication,
 - Relatively sparse use of documents.
- Agile teams, however, do follow defined and disciplined processes and carry out rigorous designs:
 - This is in contrast to chaotic coding in exploratory programming.

Extreme Programming (XP)

Extreme Programming Model

- Extreme programming (XP) was proposed by Kent Beck in 1999.
- The methodology got its name from the fact that:
 - Recommends taking the best practices to extreme levels.
 - If something is good, why not do it all the time.

Taking Good Practices to Extreme

- **If code review is good:**
 - Always review --- **pair programming**
- **If testing is good:**
 - Continually write and execute test cases ---
test-driven development
- **If incremental development is good:**
 - Come up with new increments every few days
- **If simplicity is good:**
 - Create the simplest design that will support
only the currently required functionality.

Taking to Extreme

- **If design is good,**
 - everybody will design daily (refactoring)
- **If architecture is important,**
 - everybody will work at defining and refining the architecture (metaphor)
- **If integration testing is important,**
 - build and integrate test several times a day (continuous integration)

4 Values

- **Communication:**
 - Enhance communication among team members and with the customers.
- **Simplicity:**
 - Build something simple that will work today rather than something that takes time and yet never used
 - May not pay attention for tomorrow
- **Feedback:**
 - System staying out of users is trouble waiting to happen
- **Courage:**
 - Don't hesitate to discard code

Best Practices

- **Coding:**
 - without code it is not possible to have a working system.
 - Utmost attention needs to be placed on coding.
- **Testing:**
 - Testing is the primary means for developing a fault-free product.
- **Listening:**
 - Careful listening to the customers is essential to develop a good quality product.

Best Practices

- **Designing:**
 - Without proper design, a system implementation becomes too complex
 - The dependencies within the system become too numerous to comprehend.
- **Feedback:**
 - Feedback is important in learning customer requirements.

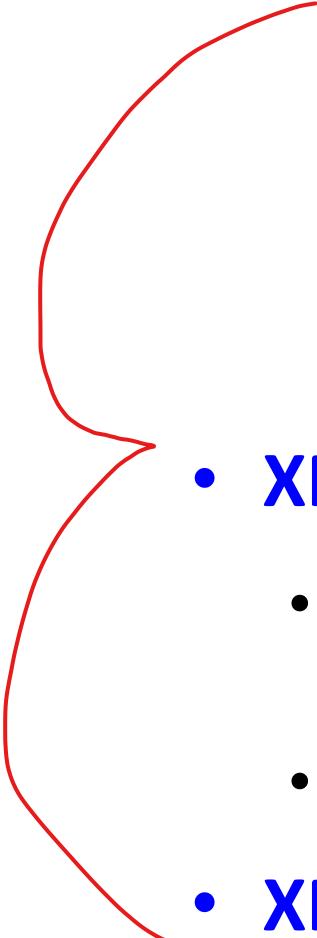
Extreme Programming Activities

- **XP Planning**

- Begins with the creation of “user stories”
- Agile team assesses each story and assigns a cost
- Stories are grouped to for a deliverable increment
- A commitment is made on delivery date

- **XP Design**

- Follows the KIS principle
- Encourage the use of CRC cards
- For difficult design problems, suggests the creation of “spike solutions”—a design prototype
- Encourages “refactoring”—an iterative refinement of the internal program design



Extreme Programming Activities

- **XP Coding**

- Recommends the construction of unit test cases *before* coding commences (test-driven development)
- Encourages “pair programming”

- **XP Testing**

- All unit tests are executed daily
- “Acceptance tests” are defined by the customer and executed to assess customer visible functionalities

1. **Planning** – determine scope of the next release by combining business priorities and technical estimates

Full List of XP Practices

2. **Small releases** – put a simple system into production, then release new versions in very short cycles
3. **Metaphor** – all development is guided by a simple shared story of how the whole system works
4. **Simple design** – system is to be designed as simple as possible
5. **Testing** – programmers continuously write and execute unit tests

Full List of XP Practices

7. **Refactoring** – programmers continuously restructure the system without changing its behavior to remove duplication and simplify
8. **Pair-programming** -- all production code is written with two programmers at one machine
9. **Collective ownership** – anyone can change any code anywhere in the system at any time.
10. **Continuous integration** – integrate and build the system many times a day – every time a task is completed.

Full List of XP Practices

- 11. 40-hour week** – work no more than 40 hours a week as a rule
- 12. On-site customer** – a user is a part of the team and available full-time to answer questions
- 13. Coding standards** – programmers write all code in accordance with rules emphasizing communication through the code

Emphasizes Test-Driven Development (TDD)

- Based on user story develop test cases
- Implement a quick and dirty feature every couple of days:
 - Get customer feedback
 - Alter if necessary
 - Refactor
- Take up next feature

Project Characteristics that Suggest Suitability of Extreme Programming

- Projects involving new technology or research projects.
 - In this case, the requirements change rapidly and unforeseen technical problems need to be resolved.
- Small projects:
 - These are easily developed using extreme programming.

Thank You!!