

# Software Engineering

# Organization of this Lecture:



- What is Software Engineering?
- Programs vs. Software Products
- Evolution of Software Engineering
- Notable Changes In Software Development Practices
- Introduction to Life Cycle Models
- Summary

# What is Software Engineering?


- Engineering approach to develop software.
  - Building Construction Analogy.
- Systematic collection of past experience:
  - techniques,
  - methodologies,
  - guidelines.
  - Counting number of persons in a room v/s the Census (i.e. population of a country )




# What do we want in a system?





- Reliability
- Efficiency
- Maintainable/ Adaptable
- Accessible / Usable (good interface design)
- Cost Effectiveness
- Safety
- Robustness
- Recoverability
- Portability
- Understandability



**Reliability** describes the ability of a system to keep functioning correctly under specific use over a given period. (System should not fail in adverse circumstances) e.g. health care system, monitoring system.

- 
- The test for **efficiency** measures the required **time** and consumption of **resources** for the fulfillment of tasks. Resources may include other software products, the software and hardware configuration of the system etc. e.g. ATM.

- 
- **Usability** is very important for **interactive** software systems. Users will not accept a system that is hard to use. So the system should be easy to use.

- 
- **Fault tolerance** is the capability of the software product to maintain a specified level of performance, or to **recover** from faults in cases of any software faults.



# Engineering Practice

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

# Why Study Software Engineering? (1)

- To acquire skills to develop **large** programs.
  - Exponential growth in complexity and difficulty level with **size**.
  - The **ad-hoc** approach breaks down when size of software increases.

## Why Study Software Engineering? (2)



- Ability to solve complex programming problems:
  - How to **break/divide** large projects into smaller and manageable parts?
- Learn techniques of:
  - specification, design, interface development, testing, project management, etc.

## Why Study Software Engineering? (3)

- To acquire skills to be a better ***software engineer*** (analyst/designer/developer/tester...) instead of a programmer:
  - \* To achieve Higher **Productivity**
  - \* Produce Better **Quality** Programs...

# Software Crisis



- In early days, Software products:
  - fail to meet user requirements.
  - frequently crash.
  - expensive.
  - difficult to alter, debug, and enhance.
  - often delivered late.
  - use resources non-optimally.

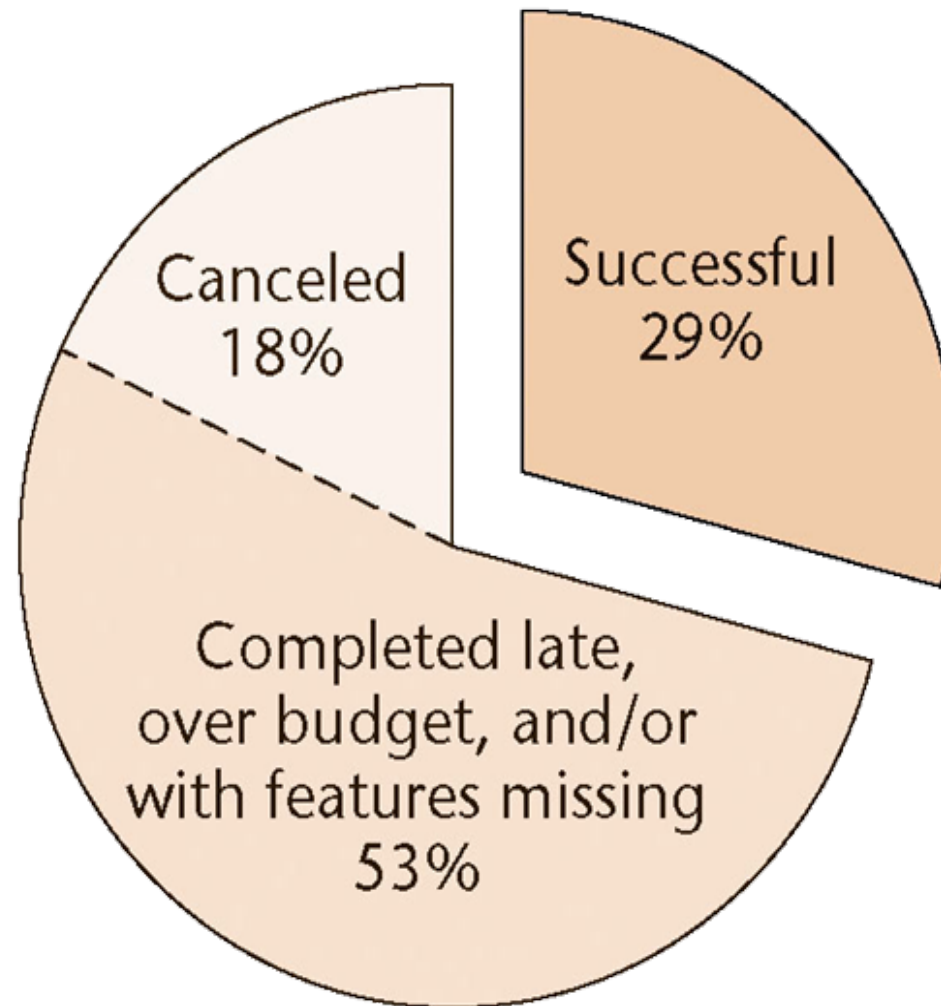
# Software Crisis...



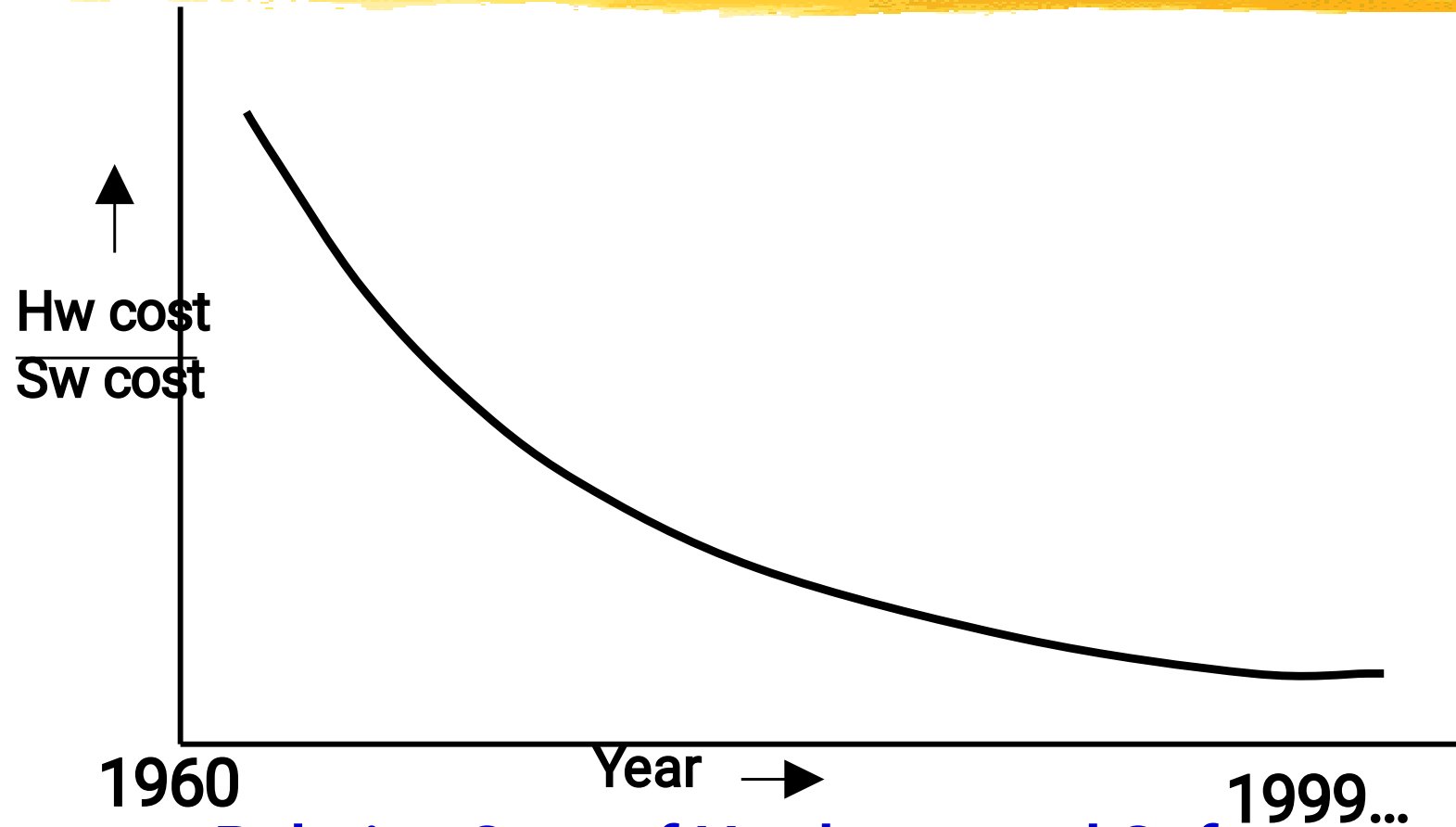
- Software projects:
  - Over budgets.
  - Over Schedules.
  - Large number of Cancellation.
  - Uncontrolled and unmanaged.

# Software Crisis...

- Research from Standish Group Data on 9236 development projects completed in **2004**.



# Software Crisis (cont.)



Relative Cost of Hardware and Software

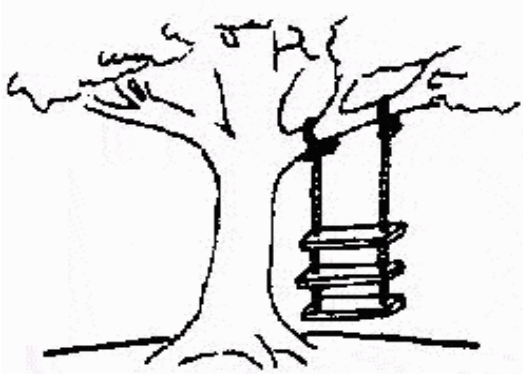


# Factors Contributing to the Software Crisis

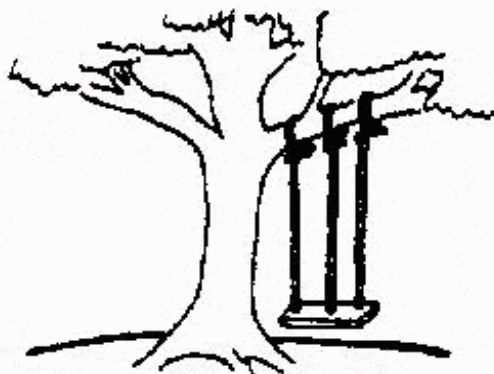


- **Larger and complex** problems,
- **Lack** of adequate training in software engineering,
- **Shortage** of skilled people,
- Low productivity and quality improvements.

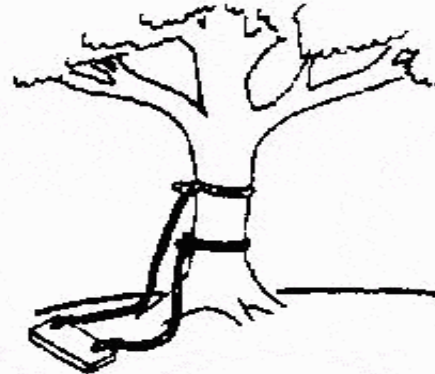
# How was Software usually Constructed ...



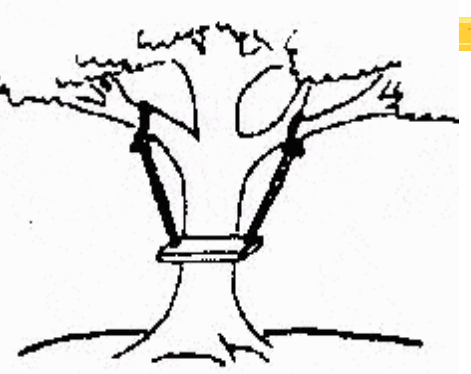
The requirements specification was defined like this



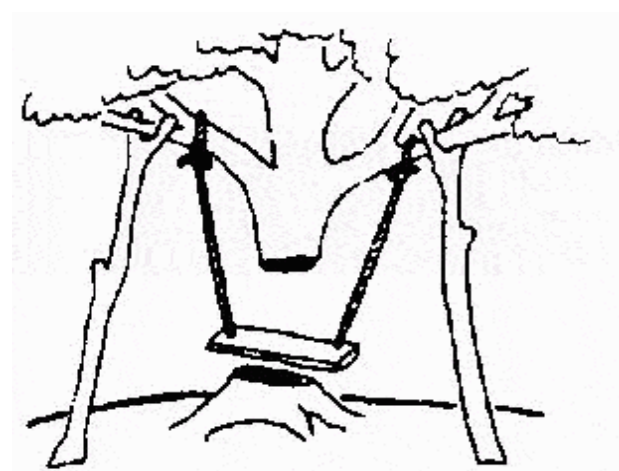
The developers understood it in that way



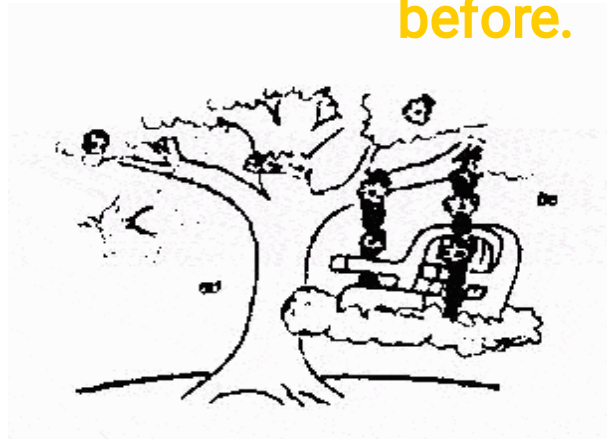
This is how the problem was designed before.



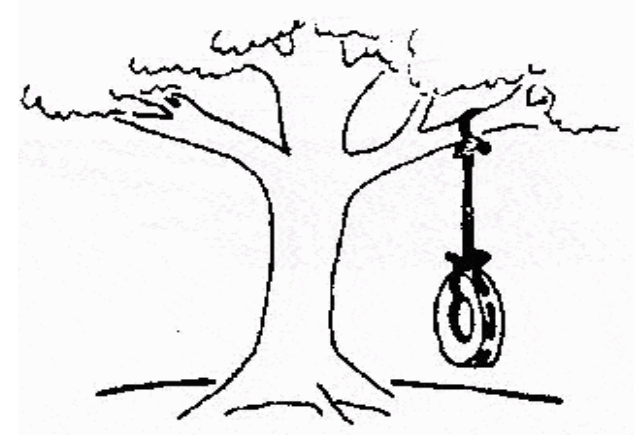
This is how the programmers coded it



That is the program after debugging



This is how the program is described by marketing dept.



This, in fact, is what the customer wanted ... :-)

# Software in the 21st Century...

- More **safety critical, real-time** software..
- **Embedded software** is everywhere ... check your **pockets**,
- Smart devices... **Intelligent systems**...
- Enterprise applications means **bigger programs, more users**
- Security is now all about software faults,
  - **Secure software** is **reliable** software.
- The **web** offers a new deployment platform,
  - Very competitive and highly available to large number of users,
  - Web apps must be highly **reliable**.

# What is Software Engineering?



IEEE :Software Engineering:

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

# Software Engineering...





- Software engineering is a discipline whose aim is the production of ***fault-free*** software, delivered ***on time and within the given budget***, that ***satisfies all the user's requirements***.

# Software Engineering...



- Broad definition:
- The methods, processes, technologies and tools by which software systems are developed and maintained.
- Encompasses many areas:
  - Planning and managing software projects
  - Requirements Analysis and Specification
  - System Modeling and Design
  - System Implementation
  - Testing and quality assurance
  - Maintenance

- 
- The important approach to tackle the complexity of given problem is **decomposition**. In this technique, a complex problem is divided into several smaller problems and then the smaller problems are solved one by one.

- 
- The important principle of **abstraction** implies that a problem can be simplified by omitting irrelevant details. In other words, the main purpose of abstraction is to consider only those aspects of the problem that are *relevant for certain purpose* and suppress other aspects that are not relevant at this time and for given purpose.



# Software Characteristics



- **Developed or Engineered – not “manufactured”**
  - Software is logical system element
  - Software projects managed differently than manufacturing projects
- **Software doesn’t physically wear out like hardware might**
- **Most software are **custom** built**

# Software...



- Computer programs and associated documentation such as **requirements**, **design models** and **user manuals**.
- Software products may be developed for a particular **customer** or may be developed for a **general** market.
- New software can be created by developing **new** programs, **configuring** generic software systems or **reusing** existing software.

# software..

*Software products may be:*

- **GENERIC**
  - developed to be sold to a range of different customers e. g. PC software such as **Excel** or **Word**.
  - referred to as commercial off-the-shelf (**COTS**) software or clickware supplied by a vendor

Personalized (**custom made**) - developed for a single customer according to their specification.

Product specification controlled by the product developer

# What are the important attributes of good software?



- The software should deliver the required **functionality** and **performance** to the user and should be maintainable, dependable and acceptable.
- Maintainability:
  - Software must evolve to **meet changing needs**;
- Dependability
  - Software must be **reliable**;
- Efficiency
  - Software should not make wasteful use of **system resources**;
- Acceptability
  - Software must **accepted by the users** for which it was designed. This means it must be understandable, usable and compatible with other systems.

# Software Characteristics

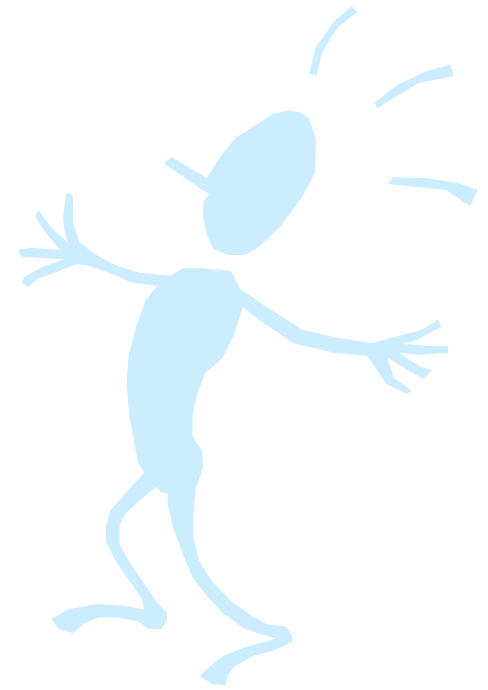


- **Many Types..**
  - System software, Real Time systems, Business software, Scientific software, Embedded software, PC based software, Web based software, AI software etc.
- **Products**
  - Generic or Packaged (COTS)
  - Custom Built
- **Sources**
  - Open (shareware) easily shared,
  - Closed (Proprietary), have to be purchased

# Software Myths (Management Perspectives)

As long as there are good standards and clear procedures in my company, I shouldn't be too concerned.

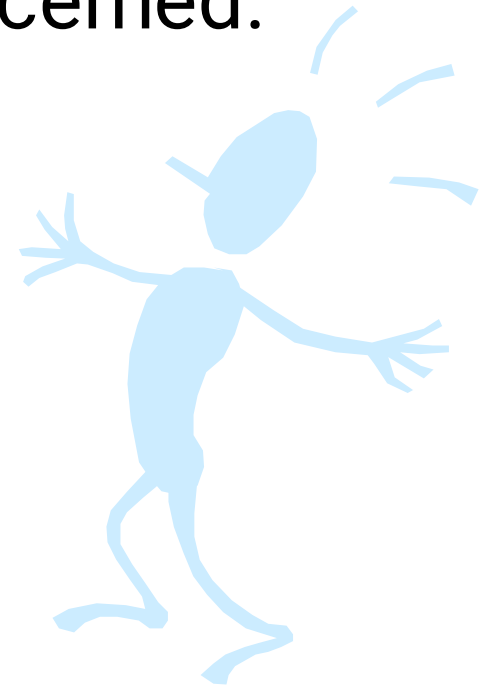
*But the proof of the pudding  
is in the eating;  
not in the Recipe !*



# Software Myths (Management Perspectives)

As long as my software engineers, have access to the fastest and the most sophisticated computer environments and state-of-the-art software tools, I shouldn't be too concerned.

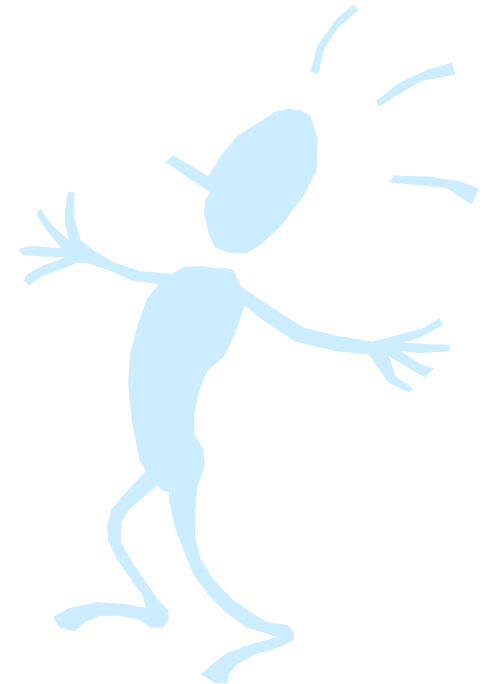
***The environment is only one of the several factors that determine the quality of the end software product!***



# Software Myths (Management Perspectives)

When my schedule slips, what I have to do is to start a fire-fighting operation: add more software specialists, those with higher skills and longer experience - they will bring the schedule back on the rails.

*Unfortunately,  
software business does not  
entertain schedule compaction  
beyond a limit!*





# Software Myths (Customer Perspectives)

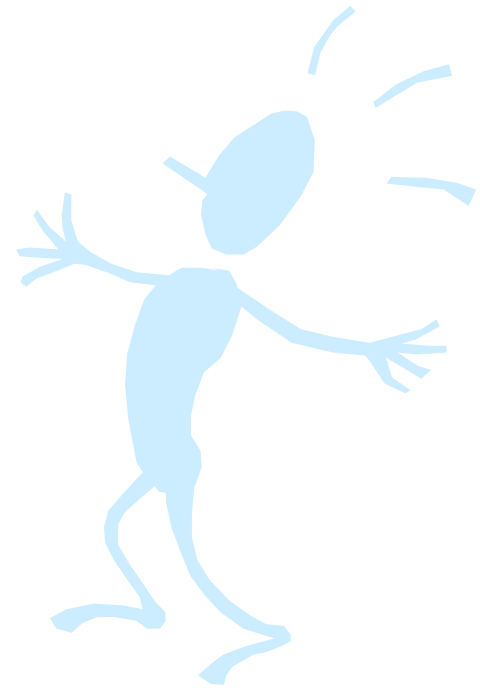
---

- A general statement of objectives is sufficient to get started with the development of software. Missing/vague requirements can easily be incorporated even late in the development process.

# Software Myths (Developer Perspectives)

Once the software is demonstrated, the job is done.

*Usually, the problems just begin!*



# Software Myths (Developer Perspectives)

Until the software is coded and is available for testing, there is no way for assessing its quality.

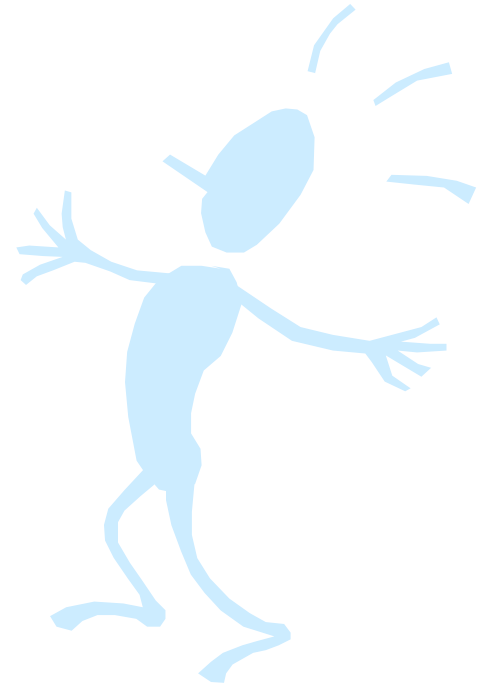
*Usually, there are too many tiny bugs inserted at every stage that grow in size and complexity as they progress thru further stages!*



# Software Myths (Developer Perspectives)

The only deliverable for a software development project is the tested code.

*The code is only  
the externally visible component  
of the entire software!*



# Software Product

---

is a product designated for delivery to the user

source  
codes

documents

reports

object  
codes

plans

manuals

data

test suites

test results

prototypes

# Boehm's Top Ten Industrial Software Metrics



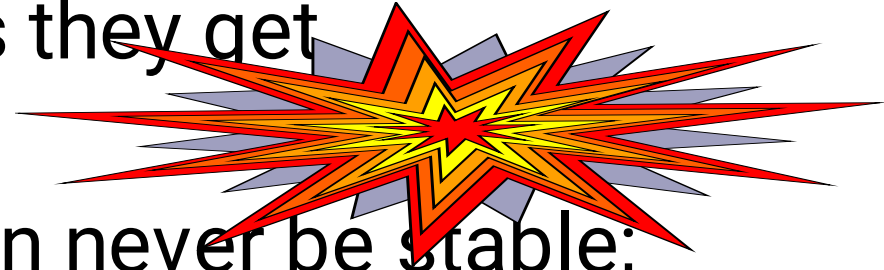
# Boehm's Top Ten Industrial Software Metrics

---

- 1 Finding and fixing a software problem after delivery of the product is 100 times more expensive than defect removal during requirements and early design phases.

# Software Myths (Customer Perspectives)

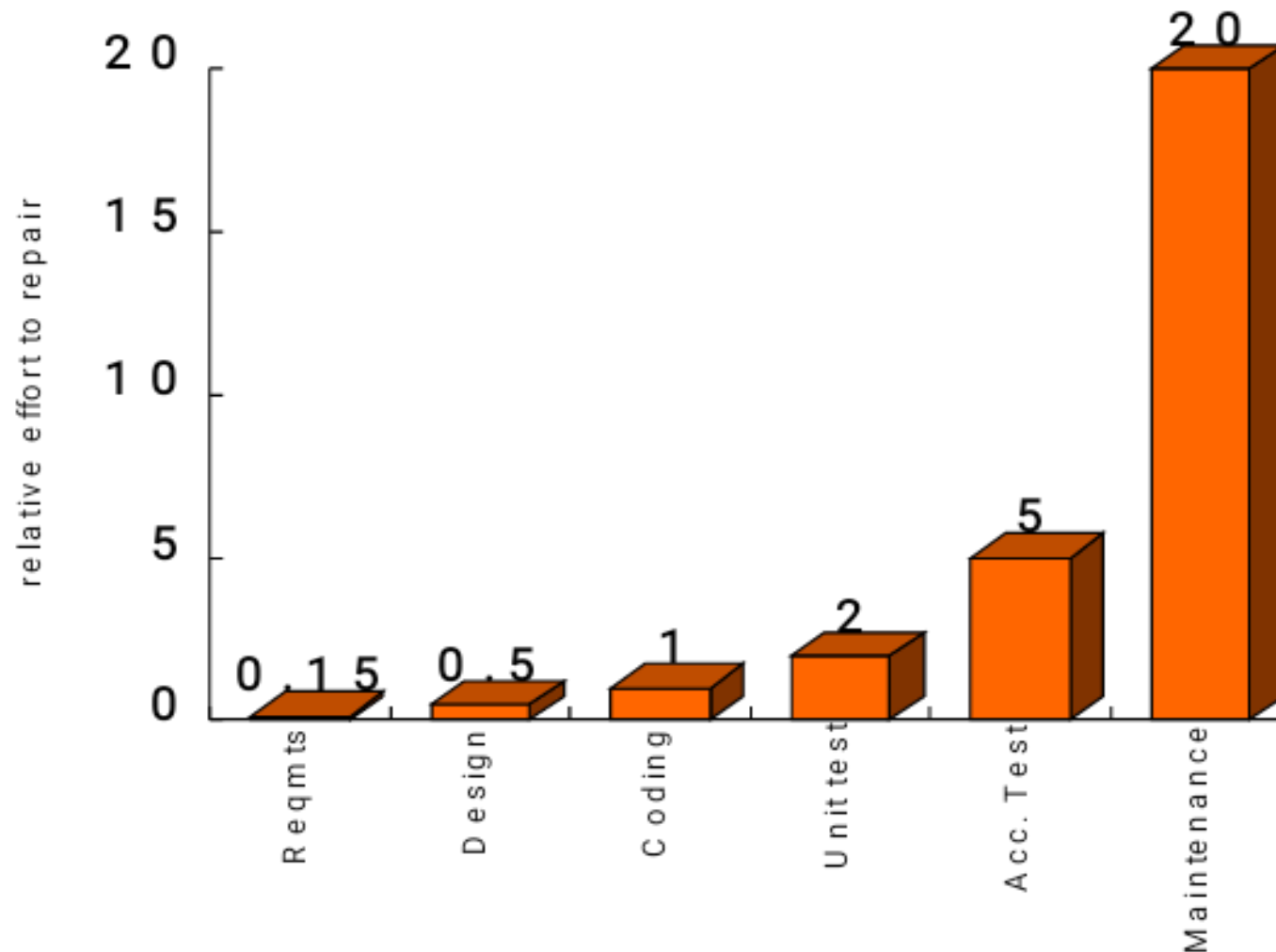
- A general statement of objectives is sufficient to get started with the development of software. Missing/vague requirements can easily be incorporated/detailed out as they get concretized.
- Application requirements can never be stable; software can be and has to be made flexible enough to allow changes to be incorporated as they happen.






# Effort to Repair Software

(when defects are detected at different stages)



- 
- Small **bug** in the initial phase (requirements),
  - Becomes an **error** in later phases,
  - May become **serious fault** in testing phase,
  - And can be the major cause of system, **failure** after implementation.

# Boehm's Top Ten Industrial Software Metrics

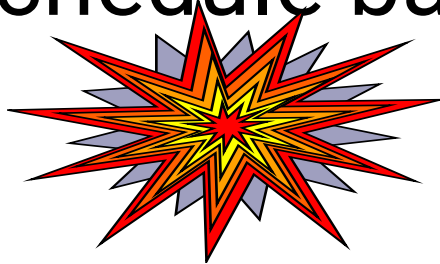
---

2 Nominal software development schedules can be compressed up to 25% (by adding people, money, etc.) but no more.

# Software Myths (Management Perspectives)

---

- When my schedule slips, what I have to do is to start a fire-fighting operation: add more software specialists, those with higher skills and longer experience - they will bring the schedule back on the rails!



# Boehm's Top Ten Industrial Software Metrics

---

3 Maintenance costs twice  
what the development costs.

# Boehm's Top Ten Industrial Software Metrics

---

4 Development and maintenance costs are primarily a function of the size.

# Boehm's Top Ten Industrial Software Metrics

---

**5** Variations in humans (people can be transferred or can change the company) account for the greatest variations in overall productivity.

# Software Myths (Management Perspectives)

---

- As long as my software engineers(!) have access to the fastest and the most sophisticated computer environments and state-of-the-art software tools, I shouldn't be too concerned.



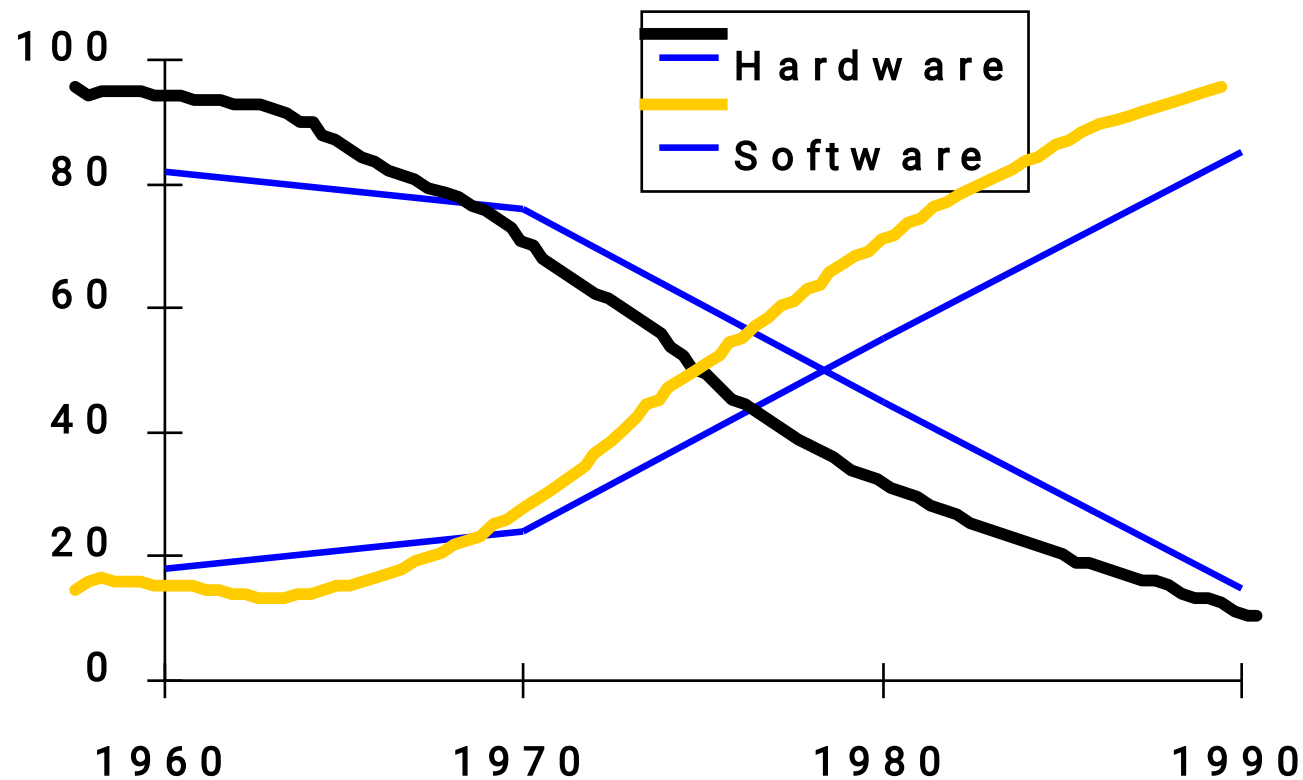


# Boehm's Top Ten Industrial Software Metrics

---

6 The ratio of software to hardware costs has gone from 15:85 in 1985 and continues to grow in favor of software as the leading cost.

# Hardware Vs Software Costs

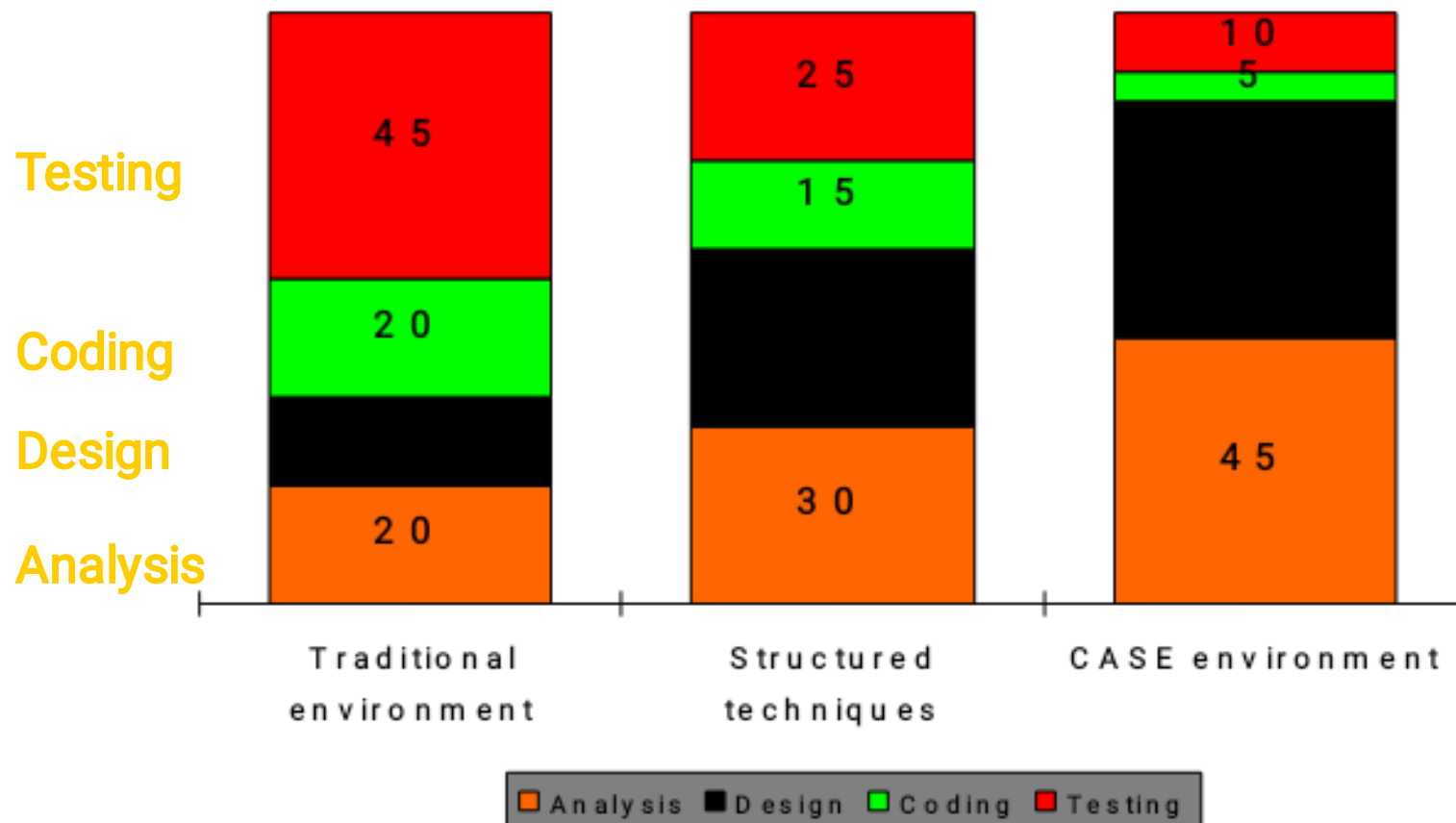


# Boehm's Top Ten Industrial Software Metrics

---

7 Only about 15% of the  
development effort is in  
coding.

# Distribution of Effort Across Phases



# What is CASE?

- **Computer-Aided Software Engineering**
- Software systems that are intended to provide **automated support** for software process activities.

## Upper-CASE

- Tools to support the early process activities of requirements and design;

- Lower-CASE

- Tools to support later activities such as programming, debugging and testing.

Reverse Engineering ??

# Boehm's Top Ten Industrial Software Metrics

---

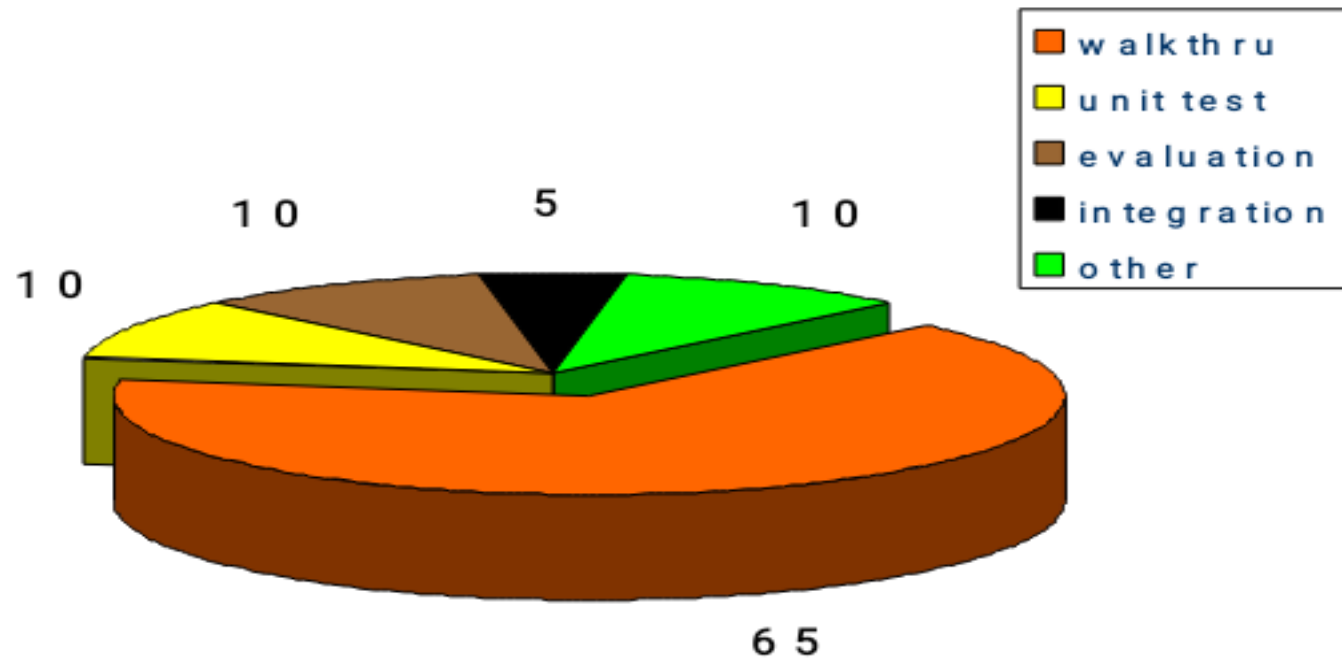
8 Application software products cost three times as much per instruction as individual programs; system software products cost nine times as much.

# Boehm's Top Ten Industrial Software Metrics

---

9 Walkthroughs/reviews catch around 65% of the errors.

# Distribution of Activities in Defect Removal



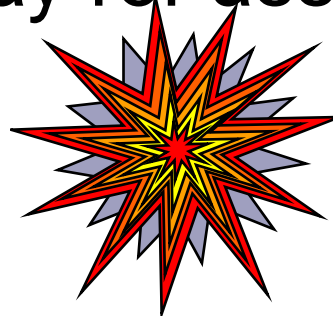


# Software Myths

## (Developer Perspectives)

---

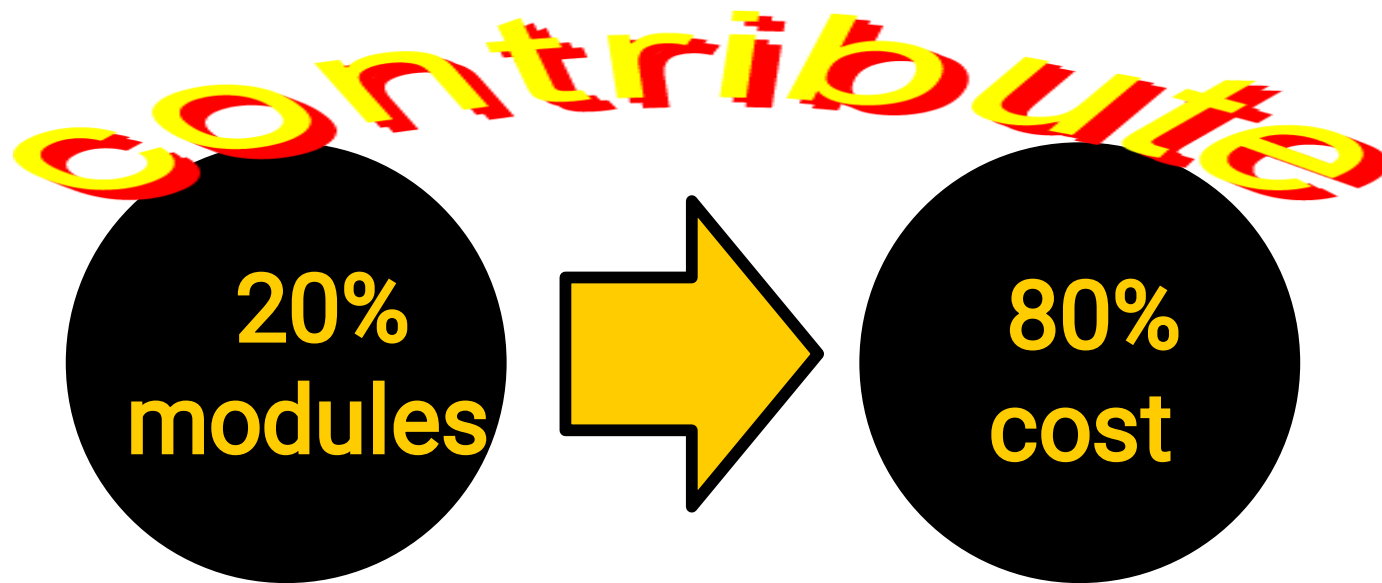
- Until the software is coded and is available for testing, there is no way for assessing its quality.



# Boehm's Top Ten Industrial Software Metrics

---

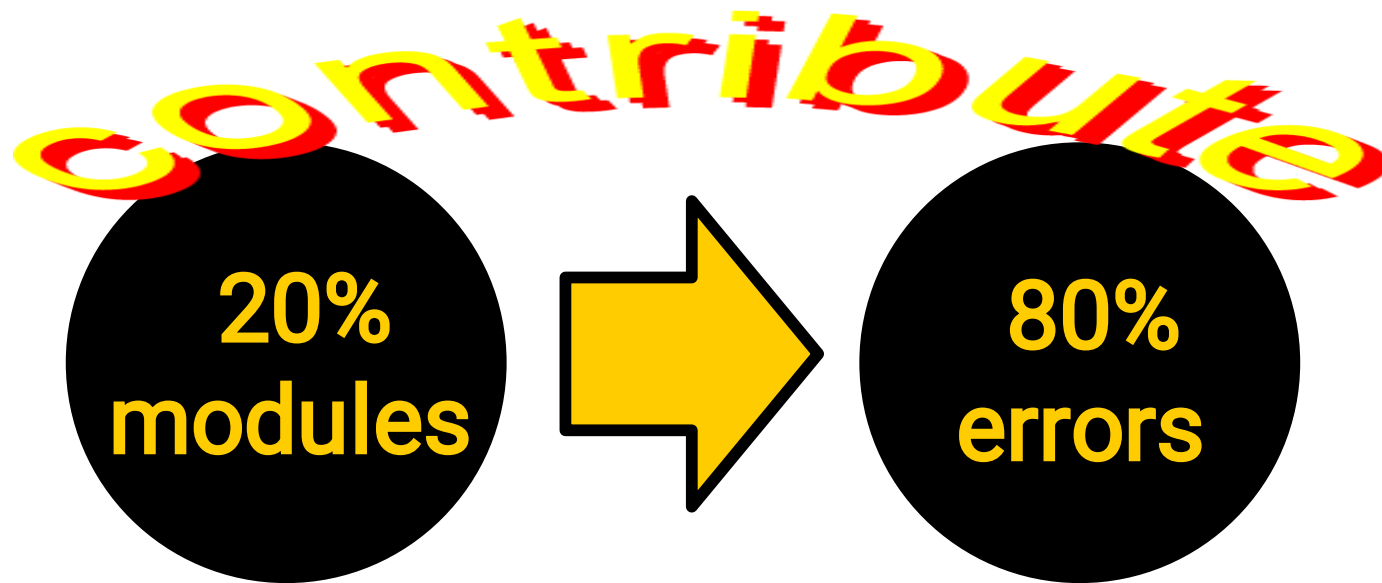
**10** Many software processes obey a Pareto distribution.



# Boehm's Top Ten Industrial Software Metrics

---

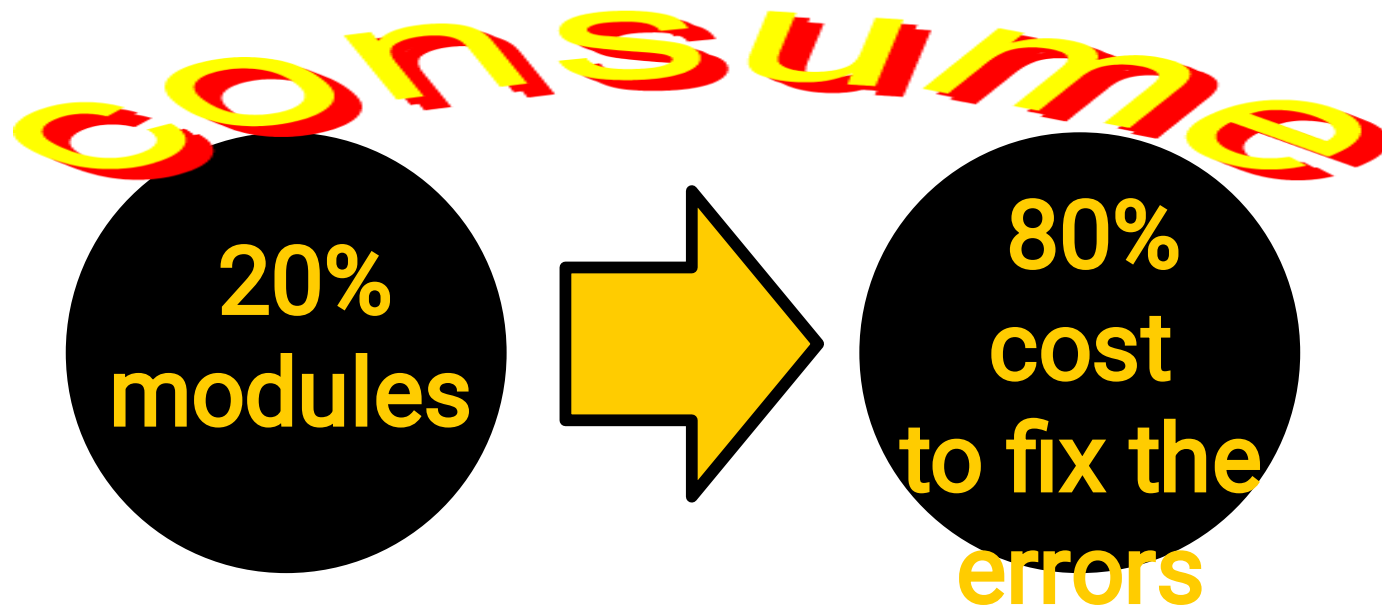
**10** Many software processes  
obey a Pareto distribution.



# Boehm's Top Ten Industrial Software Metrics

---

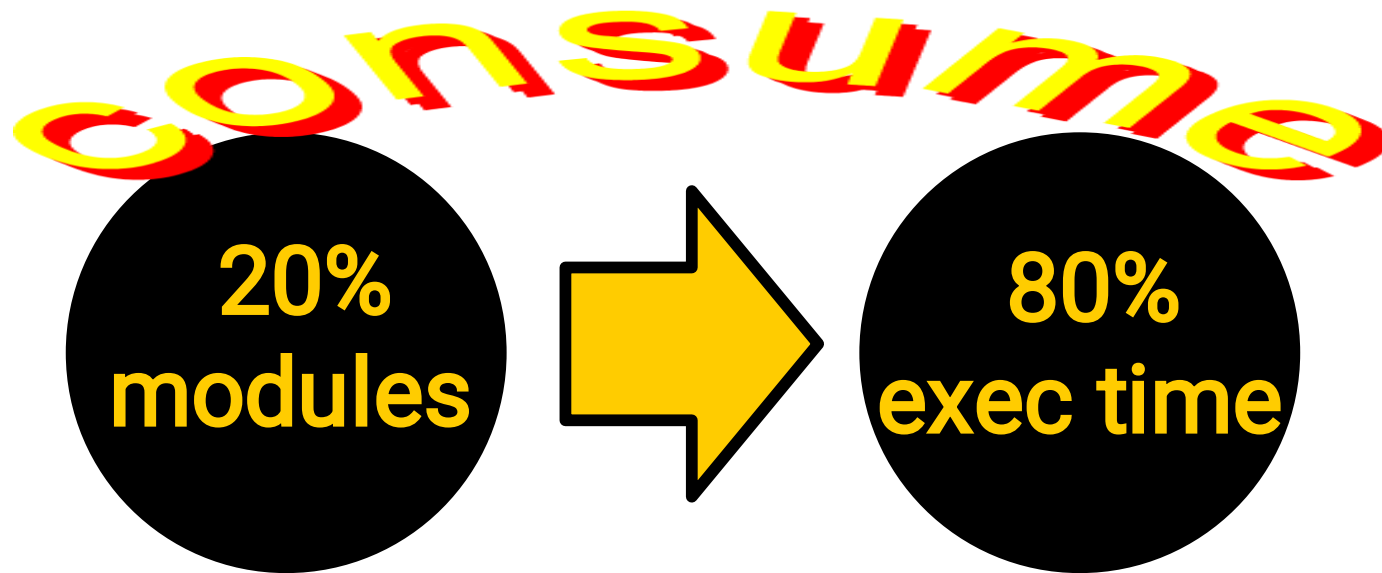
**10** Many software processes obey a Pareto distribution.



# Boehm's Top Ten Industrial Software Metrics

---

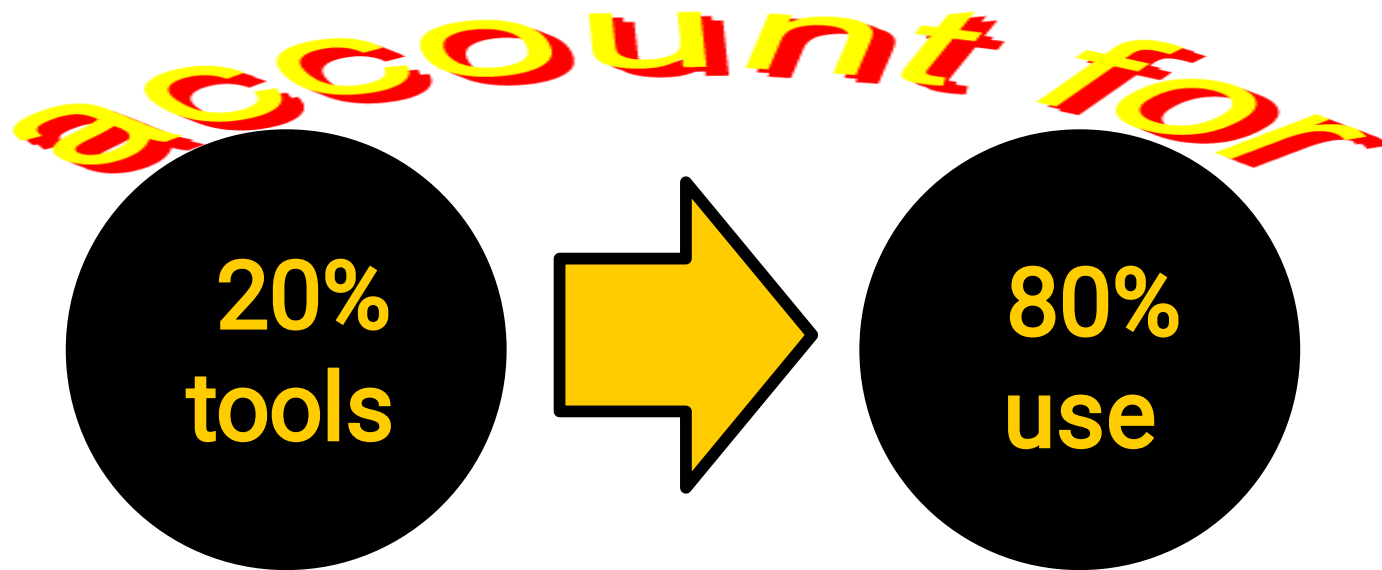
**10** Many software processes  
obey a Pareto distribution.



# Boehm's Top Ten Industrial Software Metrics

---

**10** Many software processes  
obey a Pareto distribution.



# Symptom of Software Crisis



- about US\$250 billion spent per year in the US on application development
- of this, about US\$140 billion wasted due to the projects getting discarded or reworked; this in turn because of **not following best practices and standards in the development process i.e. S.E.**

# Symptom of Software Crisis



- 10% of client/server apps are abandoned or restarted from scratch
- 20% of apps are significantly altered to avoid disaster
- 40% of apps are delivered significantly late

**Source: 3 year study of 70 large comp apps 30 European firms.  
Compuware (12/1995)**



# Programs versus Software Products

• Usually small in size	• Large
• Author himself is sole user	• Large number of users
• Single developer	• Team of developers
• Lacks proper user interface	• Well-designed interface
• Lacks proper documentation	• Well documented & user-manual prepared
• Ad hoc development.	• Systematic development

# Programs versus Software Products



- What is the difference between a student's program and industrial strength s/w for the given problem?

# Programs versus Software Products



## Student Program

- Developer is the user
  - bugs are tolerable,
  - UI not so important,
  - No documentation,
  - No thorough testing.

## Industrial Strength

- Others are the users
  - bugs not tolerated,
  - UI very imp. Issue,
  - Tested carefully,
  - Documents needed for the user as well as for the organization and the project.

# Programs versus Software Products



## Student Program

- SW not in critical use,
- Reliability, robustness not important,
- Not much investment,
- Don't care about portability.

## Industrial Strength

- Supports important functions / business,
- Reliability, robustness are very important,
- Heavy investment,
- Portability, scalability is a key issue here,

# Computer Systems Engineering



- Computer systems engineering:
  - encompasses software engineering.
- Many products require development of software as well as specific hardware to run it:
  - an ATM,
  - a coffee vending machine,
  - a mobile communication product, etc.

# Computer Systems Engineering

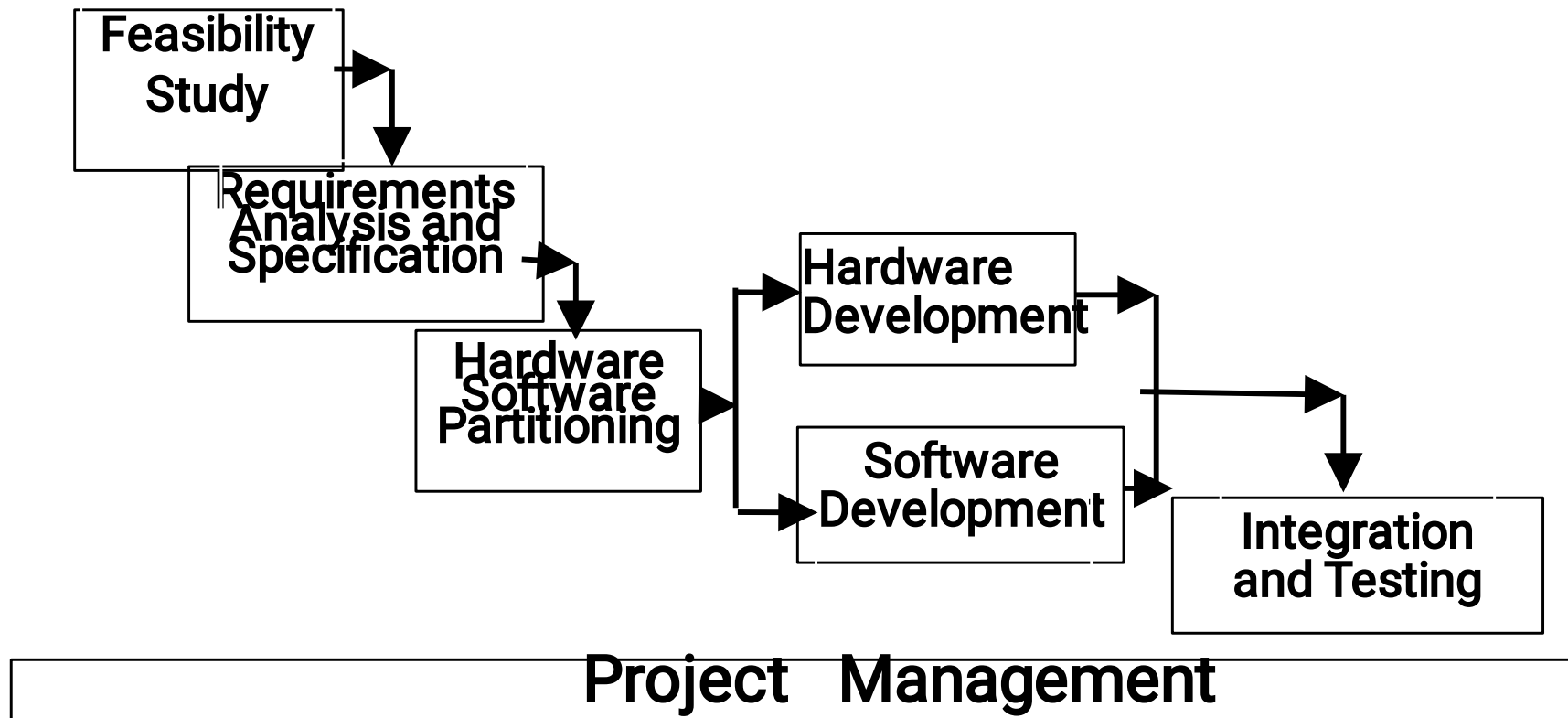


- The high-level problem:
  - deciding which tasks are to be solved by software
  - which ones by hardware.

# Computer Systems Engineering (CONT.)

- Often, 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

- Early Computer Programming (1950s):
  - Programs were being written in **m/c language** initially and then in **assembly language**.
  - Programs were 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/her convenience (called **exploratory programming/ adhoc programming**) which leads to **Adhoc development**.

# High-Level Language Programming

(Early 60s)

- Later, High-level languages such as FORTRAN, BASIC, and COBOL were introduced:
  - This reduced software development efforts greatly.

# 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.

....

in 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:
  - programs **written by others** were very difficult to understand and maintain.
- 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)



- A program's control structure indicates:
  - the sequence in which the 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 a program's control structure.
  - Usually one understands a program:
    - \* by mentally simulating the program's execution sequence.



# Control Flow-Based Design

(Late 60s)

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

# Control Flow-Based Design (Late 60s)

- After some time, It was found:
  - GO TO statements makes control structure of a program messy
  - GO TO statements alter the flow of control arbitrarily.
  - Need to **restrict** the use of GO TO statements was recognized.

# Control Flow-Based Design (Late 60s)



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

# Control-flow Based Design (Late 60s)



- At that time, **Dijkstra** published his article in a research paper:
  - “**Goto Statement Considered Harmful**” Comm. of ACM, 1969.
  - (a milestone in this journey)
- Many programmers were unhappy to read his article.

# Control Flow-Based Design (Late 60s)



- They published several counter articles:
  - highlighting the advantages and unavailability of GO TO statements.

# Control Flow-Based Design (Late 60s)

- But, Soon it was conclusively proved:
  - only three programming constructs are sufficient to express any programming logic:
    - \* **sequence** (e.g. `a=0;b=5;`)
    - \* **selection** (e.g. `if(c=true) k=5 else m=5;`)
    - \* **iteration** (e.g. `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 the following types of constructs:
    - \*sequencing,
    - \*selection,
    - \*Iteration.



# Structured programs...



- Unstructured control flows are avoided.
- Consist of a neat set of modules.
- Use single-entry, single-exit program constructs.

# Structured programs ...



- However, violations to this feature are permitted:
  - due to practical considerations such as:
    - \* premature loop exit to support exception handling.

# Structured programs ...



- Structured programs are:
  - Easier to read and understand,
  - easier to maintain,
  - require less effort and time for development.

# 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** constructs.

# Data Structure-Oriented Design (Early 70s)



- 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)

- Example of data structure-oriented design technique:
  - Jackson's Structured Programming(JSP) methodology
    - \*developed by Michael Jackson in 1970s.
  - Warnier-Orr methodology

# Data Structure Oriented Design (Early 70s)



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



# Data Structure Oriented Design (Early 70s)



- In JSP methodology:
  - a program's data structures are first designed using notations for
    - \* sequence, selection, and iteration.
  - Then data structure design is 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)

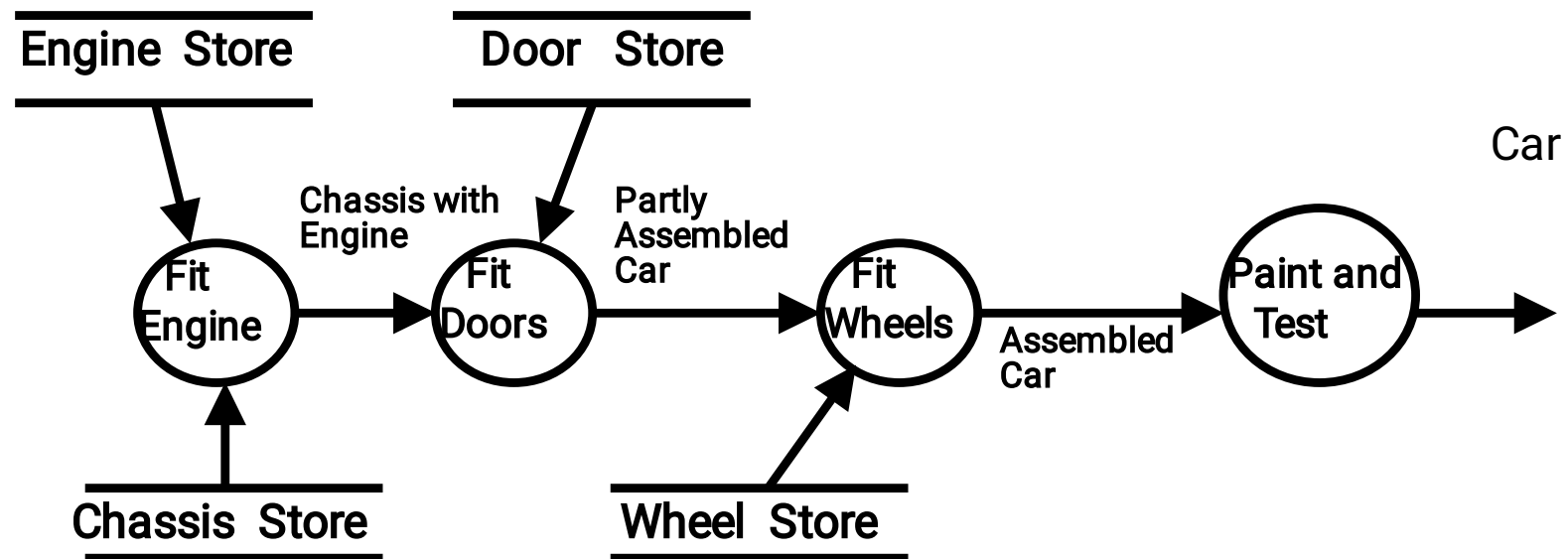


- Data flow technique identifies:
  - different processing stations (functions) in a system
  - the items (data) that flow between processing stations.

# Data Flow-Oriented Design (Late 70s)

- Data flow technique is a general 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



# Object-Oriented Design (mid 80s)



- Object-oriented technique:
  - a naturally appealing design approach:
  - natural objects (such as employees, pay-roll-register, etc.) occurring in a problem are first identified.

# Object-Oriented Design (80s)



- Relationships among objects:
  - such as composition, aggregation and inheritance are determined.
- Each object essentially acts as
  - a data hiding (or data abstraction) entity.
  - Polymorphism

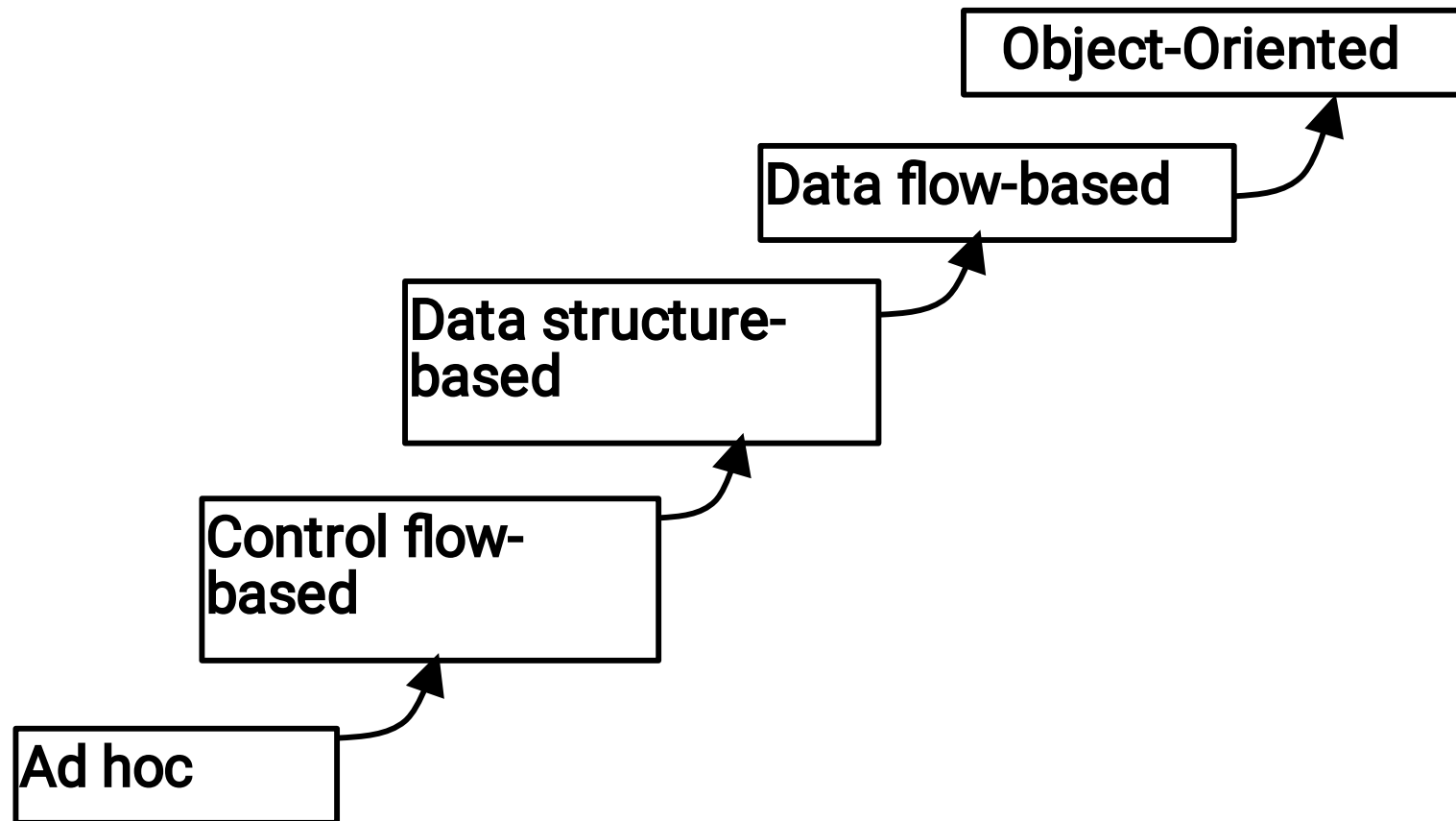


# Object-Oriented Design (80s)



- Object-Oriented Techniques have gained wide acceptance:
  - Simplicity
  - Reuse possibilities
  - Lower development time and cost
  - More robust code
  - Easy maintenance

# Evolution of Design Techniques



# Evolution of Other Software Engineering Techniques



- The improvements to the software design methodologies
  - are indeed very prominent.
- In additions to the software design techniques:
  - several other techniques evolved.

# Evolution of Other Software Engineering Techniques



- life cycle models,
- specification techniques,
- Design and modelling tools,
- project management techniques,
- testing techniques,
- debugging techniques,
- quality assurance techniques,
- software measurement techniques,
- CASE tools, etc.

# Differences Between the Exploratory Style and Modern Software Development Practices



- Use of Life Cycle Models
- Software is developed through several well-defined stages:
  - requirements analysis and specification,
  - design,
  - coding,
  - testing, etc.

# Differences Between the Exploratory Style and Modern Software Development Practices



- Emphasis has shifted
  - from error **correction** to error **prevention**.
- Modern practices emphasize:
  - detection of errors, as close as possible, to their point of introduction.
- The principle of detecting errors as close to its point of introduction as possible:
  - is known as **phase containment of errors**.

## Differences Between the Exploratory Style and Modern Software Development Practices..

---

- In exploratory style,
  - errors are detected only during testing,
- Now,
  - focus is on detecting as many errors as possible in each phase of development.

## Differences Between the Exploratory Style and Modern Software Development Practices..



- In exploratory style,
  - coding is synonymous with program development.
- Now,
  - coding is considered only a small part of overall software development effort.



# Differences Between the Exploratory Style and Modern Software Development Practices..



- Lots of effort and attention is now being paid to:
  - **requirements specification.**
- Also, now there is a distinct design phase:
  - standard design techniques are being used.

# Differences Between the Exploratory Style and Modern Software Development Practices..



- During all stages of development process:
  - **Periodic reviews** are being carried out
- Software testing has become systematic:
  - **standard testing techniques** are available.

# Differences Between the Exploratory Style and Modern Software Development Practices (CONT.)



- There is better visibility of design and code:
  - visibility means production of good quality, consistent and standard documents.
  - In the past, very little attention was being given to producing good quality and consistent documents.

## Differences between the exploratory style and modern software development practices (CONT.)



- Because of good documentation:
  - fault diagnosis and maintenance are smoother now.
- Several metrics are being used:
  - help in software project management, quality assurance, etc.

## Differences between the exploratory style and modern software development practices...



- Projects are being thoroughly planned:
  - Estimation of various resources,
  - Scheduling of different activities,
  - monitoring mechanisms.
- Use of CASE tools.

# Software Life Cycle

- 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.

# Common Symptoms of Failed Software Development Projects

- 1. Inaccurate understanding of end-user-needs
- 2. Inability to deal with changing requirements
- 3. improper designs of Modules
- 4. Software that is too hard to maintain or extend
- 5. Late discovery of serious flaws
- 6. Poor software quality
- 7. Unacceptable software performance

# Some Root Causes for Failure



- 1. Ad hoc requirements management
- 2. Ambiguous and imprecise communication between customer and developer
- 3. Poor software architectures
- 4. Overwhelming complexity
- 5. Undetected inconsistencies in requirements, design and implementations
- 6. Insufficient testing
- 7. Irregular project status assessment
- 8. Uncontrolled change propagation
- 9. Insufficient use of automation tools



# Life Cycle Model




- A software life cycle model (or process model):
  - a descriptive and diagrammatic model of software life cycle:
  - identifies all the activities required for product development,
  - establishes a prioritised ordering (sequence) among the different activities,
  - Divides life cycle into phases.

# Life Cycle Model (CONT.)




- Several different activities may be carried out in each life cycle phase.
  - For example, the design stage might consist of:
    - \* structured analysis activity followed by
    - \* Higher level design and lower level design activity.

# Why Model Life Cycle ?



- A written description:
  - forms a common understanding of activities among the software **developers**.
  - helps in identifying inconsistencies, redundancies, and omissions in the **development process**.
  - Helps in selecting a process model for specific projects (according to **application domain**).

# Why Model Life Cycle ?



- Processes are tailored for special projects.
  - A documented process model
    - \* helps to identify when and what is going to occur.

# Life Cycle Model (CONT.)



- The development team must identify a suitable life cycle model:
  - and then stick 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 ---
  - He/she has the freedom to decide his/her exact steps (when and what is to be done).

# Life Cycle Model (CONT.)



- When a software product is being developed by a team:
  - there must be a precise understanding among team members about when and what is to be done.
  - otherwise it would lead to chaos and project failure.

# Life Cycle Model (CONT.)



- 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.



# Life Cycle Model (CONT.)



- 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 (CONT.)



- It becomes easier for software project managers:
  - to monitor the progress of individual phase and overall progress of the entire project.

# Life Cycle Model (CONT.)

- When a life cycle model is adhered to,
  - the project manager can at any time accurately tell,
    - \* at which stage (e.g., design, code, test, etc. ) the project is?
  - Otherwise, 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.

# Life Cycle Model (CONT.)



- This usually leads to a problem:
  - known as the 99% complete syndrome.

# Life Cycle Model (CONT.)



- Many life cycle models have been proposed.
- We will confine our attention to a few important and commonly used models.
  - classical waterfall model
  - iterative waterfall,
  - evolutionary,
  - prototyping,
  - Agile, and
  - spiral model