# SOFTWARE ENGINEERING

SUSHREE SATAPATHY
DEPARTMENT OF COMPUTER SCIENCE
SILICON UNIVERSITY, BHUBANESWAR

# Module 1
# SOFTWARE ENGINEERING

# Syllabus

| Module | Topics |
|---|---|
| **Module – 1** | **Software Engineering: Introduction, Evolving role of software, Legacy software, Software myths, Process framework, CMM**, Life-cycle models, Waterfall model, Incremental models, Evolutionary models, Specialized models, Unified process, Agile Process Models (Extreme programming, Crystal, Scrum) |
| **Module – 2** | Requirements Engineering: Types of Requirements, Functional and non-functional requirements, The software requirements document, Requirements - specification, engineering processes, elicitation & analysis, validation, and management; Decision Trees and Decision Tables, Formal Specification (Axiomatic specs for Stacks & Queues) |
| **Module – 3** | Software Project Management: Software project planning process, Project estimation (Cost, Time, Effort), Decomposition techniques, Empirical estimation models, The Make/Buy decision, Project scheduling, Task network, Critical Path method, PERT Scheduling, Earned Value analysis |
| **Module – 4** | Design Engineering: Function-oriented Software Design (DFD Structure charts), Object-oriented Design using UML, User Interface design; Software Testing: Testing strategies, Types of testing, Black-Box testing , White-box testing, Basis Path testing, Control Structure testing, Reliability testing, Security testing. |
| **Module – 5** | Advanced Topics: Testing web-apps, Formal methods, Risk Management, Configuration Management, Re-Engineering Security Engineering. |

# Introduction

- In today's world computer software is the single most important technology in all fields.
- Every where we find computers and a controlling software.
- Incorporated in all kinds of system
  - Transportation
  - Medical
  - Telecommunication
  - Military
  - Industry
  - Education

# Introduction

- A software has a dual role
  - It is a product
  - It is a vehicle for delivering a product
- In any system, the software represents a large proportion of the total cost of the system.
- These software need to corrected , adapted and enhanced with time to suit the changing needs of the users.
- The maintenance activities  take more people and resource than to create one.

# Introduction

- Thus, given the growing need and importance of software, there are various technologies that make is easier , faster and less expensive to build and maintain high quality software.

- And here comes the role of **software engineering.**

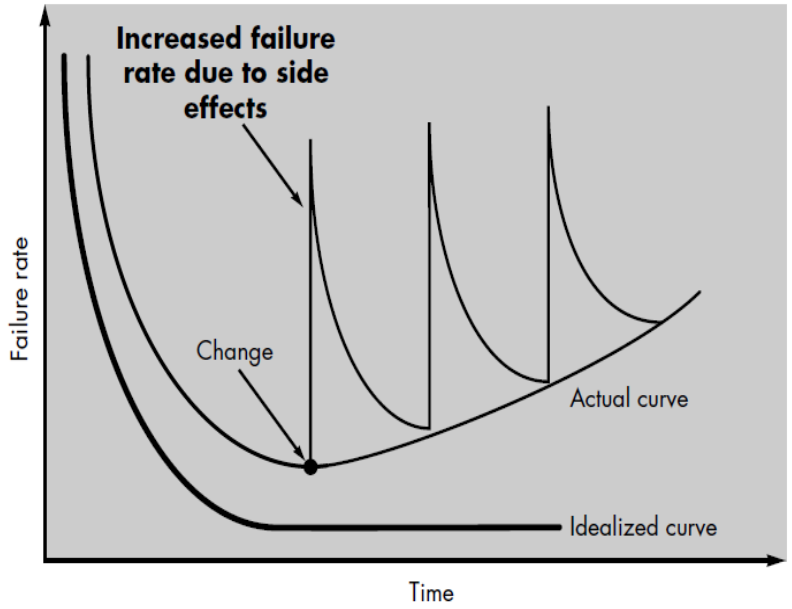
# Definitions

- **Program** : a set of instructions given to a computer to perform a particular task.
- **Software** : the collection of computer programs, procedures, rules, and associated documentation and data. (IEEE Definition)
- This implies that the discipline dealing with the development of software should not deal only with developing programs, but with developing all the things that constitute software.
- Thus, **software engineering** is largely concerned with the development of industrial-quality software.

# Software

Characteristics that differentiate software from hardware

- Software is developed or engineered; it is not manufactured in the classical sense
- Software does not wear out. It deteriorates.
- Software is intangible.




**Failure curve for software**

# Software

Characteristics that differentiate software from hardware

- Although the industry is moving toward component-based assembly, most software continues to be custom built. A software component should be designed and implemented so that it can be reused in many different programs. For example, today's graphical user interfaces are built using reusable components that enable the creation of graphics windows, pull-down menus, and a wide variety of interaction mechanisms. The data structure and processing detail required to build the interface are contained within a library of reusable components for interface construction.
- Software development is a complex process. The specification , design and testing are the difficult part of the process.

# Program vs software product

| BASIS OF COMPARISON | PROGRAM | SOFTWARE PRODUCT |
|---|---|---|
| **Developed by** | Individuals | Software professionals |
| **Number of developers** | Single developer | Large number of developers are involved |
| **Purpose** | Personal use | To be used by a large number of individuals called as customers |
| **Size** | small | Extremely large |
| **Functionality** | limited | Complex |

# Program vs software product

| BASIS OF COMPARISON | PROGRAM | SOFTWARE PRODUCT |
|---|---|---|
| **Developing strategy** | According to the programmer's individual style of development | Must be developed using accepted software engineering principles |
| **User interface** | Not very important as the programmer is the sole user and he has developed it | Must be carefully designed and implemented because developers of that product and the users of the product are totally different |
| **Documentation** | Very little documentation is expected | Software product must be well documented |

# Definition

- **Industrial strength software**
- In software engineering we are not dealing with programs that are build for demonstration purpose.
- Rather we are building software that will solve real problems of any organization and can lead to significant direct or indirect loss.
- These software are referred to as industrial strength software.

# Definition

- Characteristics of Industrial strength software

1. Expensive - very expensive because the development process is very labor intensive
2. Tight schedule – there is a demand for short delivery time for a software. Cycle time is an important driving force.
3. Productivity – high productivity leads to low cost and low cycle time. It can be measured in terms of output per unit effort(LOC per person per month)
4. Quality – developing high-quality software is the basic goal
5. Large scale – usually build for a complete business organization and needs to control everything.

# Student system vs Industrial strength software

| | STUDENT SYSTEM | INDUSTRIAL STRENGTH SOFTWARE |
|---|---|---|
| **Developer** | Developer is the user | Developer and the users are different |
| **Bugs** | Bugs are tolerable | Bugs are not tolerated |
| **User interface** | UI not important | UI is very important |
| **Documentation** | No or minimal documentation required | Documentation are required for the user as well as the organization and the project |

# Student system vs Industrial strength software

| | STUDENT SYSTEM | INDUSTRIAL STRENGTH SOFTWARE |
|---|---|---|
| **Use** | Does not have a critical use | Supports important business function |
| **Reliability** | Reliability and robustness are not important | Reliability and robustness are very important |
| **Investment** | No investment required | Heavy investment are required |
| **Portability** | Don’t care about portability | Portability is a key issue here. |

# Student system vs Industrial strength software

- Thus, the key difference lies in **quality** along with usability, reliability, portability, robustness etc.
- The **cost** and **schedule** are another two major driving forces in the development process of a software.
- Brooks thumb-rule: Industrial strength s/w costs 10 times more than student s/w
- And the area of software engineering focuses on development of such software and all the associated things that constitute a **SOFTWARE**

# Definition

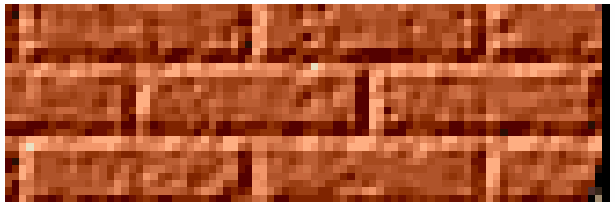
- *Software engineering* is defined as [IEE87]:
- *Software engineering* is the systematic approach to the development, operation, maintenance, and retirement of software.
- Another definition from the economic and human perspective is given by Boehm[Boe81]
- *Software Engineering* is the application of science and mathematics by which the capabilities of computer equipment are made useful to man via computer programs, procedures, and associated documentation.

# Scope and necessity of Software Engineering

- Software engineering is an engineering approach for software development.
- It is a systematic collection of past experience arranged in the form of methodologies and guidelines.
- It provides methodologies for developing software as close to the scientific method as possible. That is, these methodologies are repeatable, and if the methodology is applied by different people, similar software will be produced.

# Scope and necessity of Software Engineering

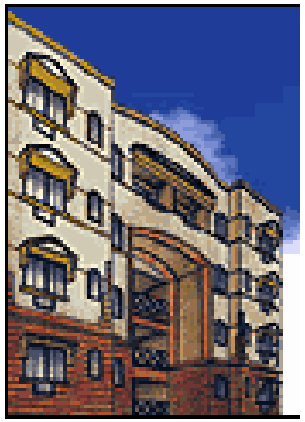
- A small program can be written without using software engineering principles. But if one wants to develop a large software product, then software engineering principles are indispensable to achieve a good quality software cost effectively.
- This can be explained using a building construction analogy.
- Suppose you want to build a small wall



- Using your common-sense you will bring bricks and cement and build a small wall very easily.

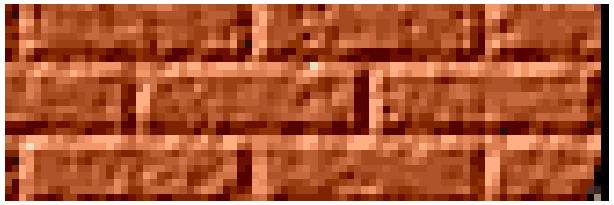
# Scope and necessity of Software Engineering

- But what if you were to build a multi-storied building..??



- Even if you tried to build a large building, it would collapse because you would not have the requisite knowledge about the strength of materials, testing, planning, architectural design, etc. to build a multi storied building.

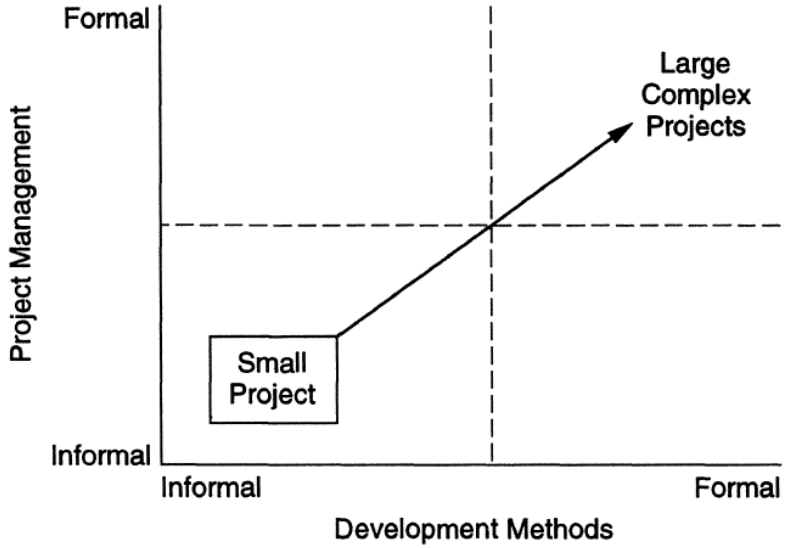
# Scope and necessity of Software Engineering



- It would be very difficult to extend your idea about a small wall construction into constructing a large building. You can use your intuition and still be successful in building a small wall, but building a large building requires knowledge of civil, architectural and other engineering

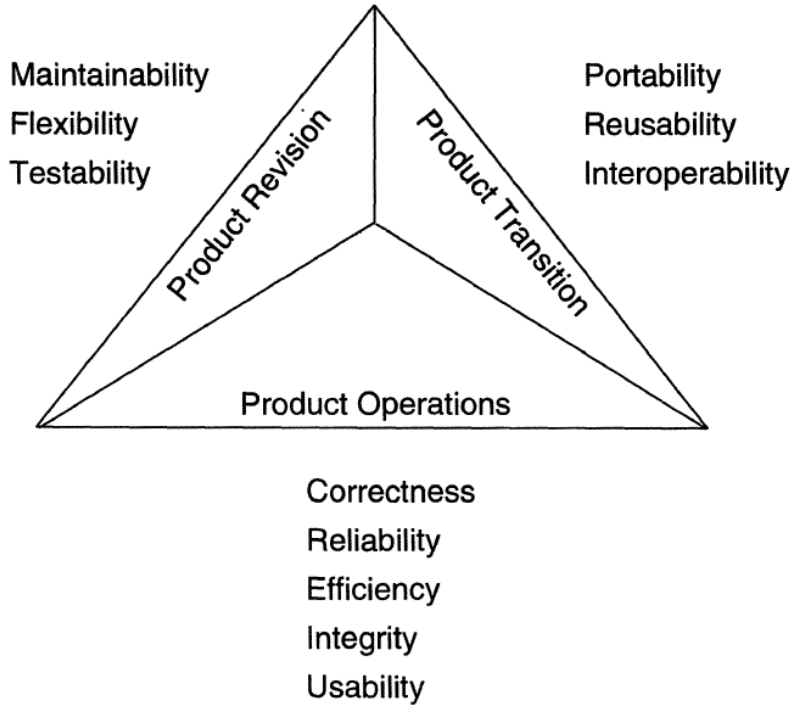
# Scope and necessity of Software Engineering

- **Challenges that software engineering faces:**
- **The problem of scale**



# Scope and necessity of Software Engineering

- **Challenges that software engineering faces:**
- **Cost, schedule and quality**
- Low cost – high quality software
- Small cycle time from concept to delivery
- Quality can be viewed as having 3 dimensions
  - Product operation,
  - Product transition and
  - Product revision.



# Scope and necessity of Software Engineering

- **Correctness** is the extent to which a program satisfies its specifications.
- **Reliability** is the property that defines how well the software meets its requirements.
- **Efficiency** is a factor in all issues relating to the execution of software; it includes such considerations as response time, memory requirement, and throughput.
- **Usability**, or the effort required to learn and operate the software properly, is an important property that emphasizes the human aspect of the system.

# Scope and necessity of Software Engineering

- **Maintainability** is the effort required to locate and fix errors in operating programs.
- **Testability** is the effort required to test to ensure that the system or a module performs its intended function.
- • **Flexibility** is the effort required to modify an operational program (perhaps to enhance its functionality).

# Scope and necessity of Software Engineering

- **Portability** is the effort required to transfer the software from one hardware configuration to another.
- **Reusability** is the extent to which parts of the software can be reused in other related applications.
- **Interoperability** is the effort required to couple the system with other systems.

# Scope and necessity of Software Engineering

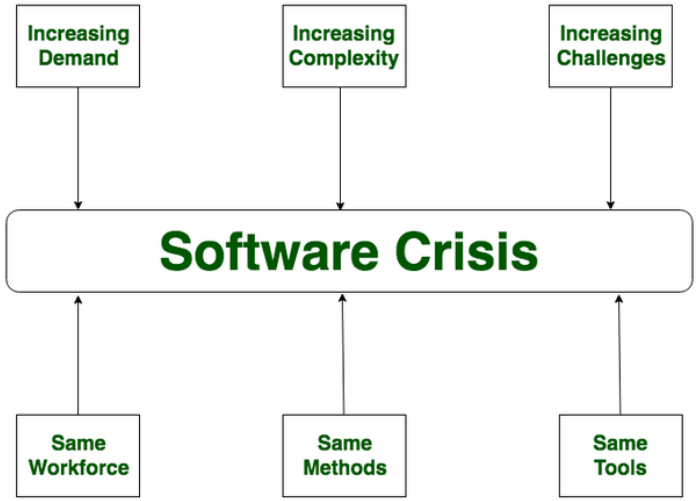
- Challenges that software engineering faces:
- The problem of consistency
  - Software development organization would like to produce consistent quality with consistent productivity.
  - Consistency allows an organization to predict the outcome of a project with reasonable accuracy, and to improve its processes to produce higher-quality products and to improve its productivity.

# Objective

- Develop methods and procedures for software development that can scale up for large systems and that can be used to consistently produce high-quality software at low cost and with a small cycle time(optimality).
- That is, the key objectives are
  - **consistency,**
  - **Low cost,**
  - **High quality,**
  - **Small cycle time, and**
  - **Scalability(applicable to large projects).**
- The basic approach that software engineering takes is to separate the development process from the developed product

# Software Crisis

- The software crisis was due to using the same workforce, same methods, and same tools even though rapidly increasing software demand, the complexity of software, and software challenges. With the increase in software complexity, many software problems arose because existing methods were insufficient.
- Organizations are appending larger portion of their budget on the software.



# Software Crisis

- Causes of software crisis
  - More expensive than the hardware
  - Use resources non-optimally
  - Often fail to meet the user requirements (low quality)
  - Frequently crash (not reliable)
  - Delivered late and overshoot the schedule
  - Difficult to debug, alter and enhance ( low maintainability)

# Software Crisis

- Factors contributing to the software crisis
  - Larger and complex problems
  - Lack of adequate training in software engineering
  - Increasing skill shortage
  - Low productivity improvements

# Software Crisis

- Solution to the software crisis
  - It is believed that the only satisfactory solution to the present software crisis can possibly come from a spread of software engineering practices among the engineers, coupled with further advancements to the software engineering discipline itself.

  - Reduction in software over budget.
  - The quality of the software must be high.
  - Less time is needed for a software project.
  - Experienced and skilled people working on the software project.
  - Software must be delivered.
  - Software must meet user requirements.

# Software Myths

- **Management Myth**

  Managers with software responsibility are often under pressure to maintain budgets, keep schedules from slipping, and improve quality.

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

- ***Reality:*** The book of standards may very well exist, but is it used? Are software practitioners aware of its existence? Does it reflect modern software engineering practice? Is it complete? Is it streamlined to improve time to delivery while still maintaining a focus on quality? In many cases, the answer to all of these questions is "no."

# Software Myths

- **Management Myth**
- ***Myth:*** My people have state-of-the-art software development tools, after all, we buy them the newest computers.
- ***Reality:*** It takes much more than the latest model mainframe, workstation, or PC to do high-quality software development. Computer-aided software engineering (CASE) tools are more important than hardware for achieving good quality and productivity, yet the majority of software developers still do not use them effectively.

# Software Myths

- **Management Myth**
- ***Myth:*** If we get behind schedule, we can add more programmers and catch up (sometimes called the *Mongolian horde concept*).
- ***Reality:*** Software development is not a mechanistic process like manufacturing. In the words of Brooks : "adding people to a late software project makes it later." At first, this statement may seem counterintuitive. However, as new people are added, people who were working must spend time educating the newcomers, thereby reducing the amount of time spent on productive development effort. People can be added but only in a planned and well-coordinated manner.

# Software Myths

- **Management Myth**
- ***Myth:*** If I decide to outsource the software project to a third party, I can just relax and let that firm build it.
- ***Reality:*** If an organization does not understand how to manage and control software projects internally, it will invariably struggle when it outsources software projects.

# Software Myths

- **Customer Myth**

A customer who requests computer software may be a person at the next desk, a technical group down the hall, the marketing/sales department, or an outside company that has requested software under contract. In many cases, the customer believes myths about software because software managers and practitioners do little to correct misinformation. Myths lead to false expectations (by the customer) and ultimately, dissatisfaction with the developer.

# Software Myths

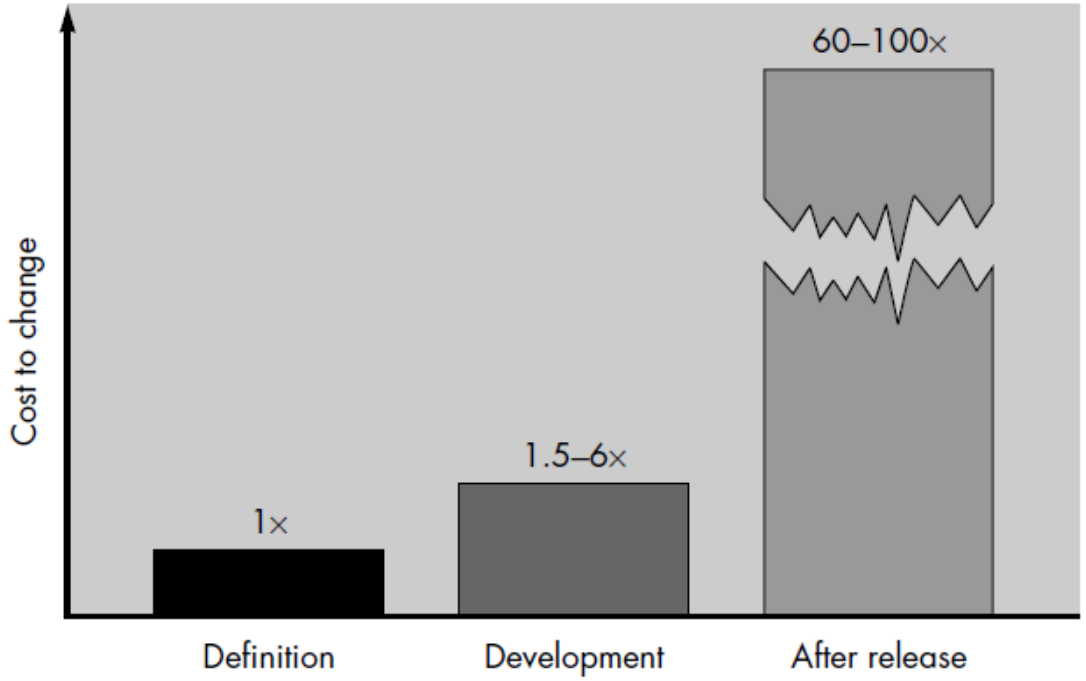
- Customer Myth
- ***Myth:*** A general statement of objectives is sufficient to begin writing programs—we can fill in the details later.
- ***Reality:*** A poor up-front definition is the major cause of failed software efforts. A formal and detailed description of the information domain, function, behavior, performance, interfaces, design constraints, and validation criteria is essential. These characteristics can be determined only after thorough communication between customer and developer.

# Software Myths

- Customer Myth
- ***Myth:*** Project requirements continually change, but change can be easily accommodated because software is flexible.
- ***Reality:*** It is true that software requirements change, but the impact of change varies with the time at which it is introduced. If serious attention is given to up-front definition, early requests for change can be accommodated easily. The customer can review requirements and recommend modifications with relatively little impact on cost. When changes are requested during software design, the cost impact grows rapidly. Resources have been committed and a design framework has been established. Change can cause upheaval that requires additional resources and major design modification, that is, additional cost. Changes in function, performance, interface, or other characteristics during implementation (code and test) have a severe impact on cost. Change, when requested after software is in production, can be over an order of magnitude more expensive than the same change requested earlier.

# Software Myths

- Customer Myth
- Figure illustrates the impact of change.



# Software Myths

- **Practitioner's Myth**

Myths that are still believed by software practitioners have been fostered by 50 years of programming culture. During the early days of software, programming was viewed as an art form.

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

# Software Myths

- Practitioner's Myth
- ***Myth:*** Until I get the program "running" I have no way of assessing its quality.
- ***Reality:*** One of the most effective software quality assurance mechanisms can be applied from the inception of a project—the *formal technical review.* Software reviews are a "quality filter" that have been found to be more effective than testing for finding certain classes of software defects.

# Software Myths

- **Practitioner's Myth**
- ***Myth:*** The only deliverable work product for a successful project is the working program.
- ***Reality:*** A working program is only one part of a *software configuration* that includes many elements. Documentation provides a foundation for successful engineering and, more important, guidance for software support.

# Software Myths

- Practitioner's Myth
- ***Myth:*** Software engineering will make us create voluminous and unnecessary documentation and will invariably slow us down.
- ***Reality:*** Software engineering is not about creating documents. It is about creating quality. Better quality leads to reduced rework. And reduced rework results in faster delivery times.

# Evolution of software design techniques over the last 50 years

- During the 1950s, most programs were being written in assembly language. These programs were limited to about a few hundreds of lines of assembly code, i.e. were very small in size. Every programmer developed programs in his own individual style - based on his intuition. This type of programming was called Exploratory Programming.

# Evolution of software design techniques over the last 50 years

- The next significant development which occurred during early 1960s in the area computer programming was the high-level language programming. Use of high-level language programming reduced development efforts and development time significantly. Languages like FORTRAN, ALGOL, and COBOL were introduced at that time.

# Evolution of software design techniques over the last 50 years

- As the size and complexity of programs kept on increasing, the exploratory programming style proved to be insufficient. Programmers found it increasingly difficult not only to write cost-effective and correct programs, but also to understand and maintain programs written by others.
- To cope with this problem, experienced programmers advised other programmers to pay particular attention to the design of the program's control flow structure (in late 1960s). It was found that the "GOTO" statement made control structure of a program complicated and messy. At that time most of the programmers used assembly languages extensively. They considered use of "GOTO" statements in high-level languages were very natural and they did not really accept that they can write programs without using GOTO statements, and considered the frequent use of GOTO statements inevitable.

# Evolution of software design techniques over the last 50 years

- At this time, Dijkstra [1968] published his (now famous) article "GOTO Statements Considered Harmful". Expectedly, many programmers were enraged to read this article. They published several counter articles highlighting the advantages and inevitably of GOTO statements. But, soon it was conclusively proved that only three programming constructs – sequence, selection, and iteration – were sufficient to express any programming logic. This formed the basis of the structured programming methodology.

# Evolution of software design techniques over the last 50 years

- **Structured programming methodology**
- A structured program uses three types of program constructs
  - ✓Selection (if, switch)
  - ✓Sequence(statements, blocks)
  - ✓Iteration(loops like for , while)
- Structured programs avoid unstructured control flows by restricting the use of GOTO statements.
- A structured program consists of a well partitioned set of modules.
- Structured programming uses single entry, single-exit program constructs such as if-then-else, do-while, etc.
- Thus, the structured programming principle emphasizes designing neat control structures for programs.

# Evolution of software design techniques over the last 50 years

- The next important development was data structure-oriented design. Programmers argued that for writing a good program, it is important to pay more attention to the design of data structure of the program rather than to the design of its control structure. Data structure-oriented design techniques actually help to derive program structure from the data structure of the program.

# Evolution of software design techniques over the last 50 years

- Next significant development in the late 1970s was the development of data flow-oriented design technique. Experienced programmers stated that to have a good program structure, one has to study how the data flows from input to the output of the program. Every program reads data and then processes that data to produce some output. Once the data flow structure is identified, then from there one can derive the program structure.

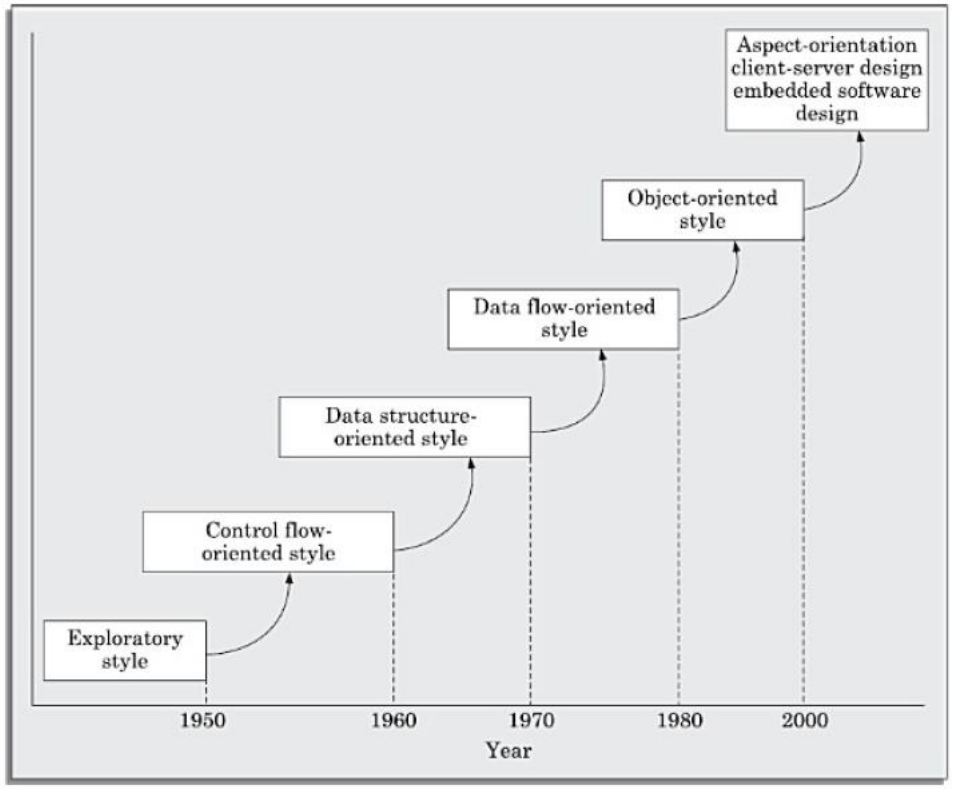
# Evolution of software design techniques over the last 50 years

- Object-oriented design (1980s) is the latest and very widely used technique. It has an intuitively appealing design approach in which natural objects (such as employees, pay-roll register, etc.) occurring in a problem are first identified. Relationships among objects (such as composition, reference and inheritance) are determined. Each object essentially acts as a data hiding entity. The program structure here is determined based on the interactions that occurs between the objects.

# Evolution of software design techniques over the last 50 years

- What next?
- Current challenges in designing software
  - Program size is increasing
  - Software are required to work in client-server environment accessed through a web browser
  - Rapid growth in embedded devices
- Solution : development of applications for hand-held devices and embedded processors.
- This requires aspect-oriented programming, client-server based design and embedded software design.

# Evolution of software design techniques over the last 50 years



# Exploratory style vs Modern style of software development

| | EXPLORATORY STYLE | MODERN STYLE |
|---|---|---|
| **Development strategy** | No fixed strategy | Development goes through several well-defined stages such as requirement specification, design, coding, testing etc. |
| **Error removal** | Error correction | Error prevention(phase containment of errors) |
| **Coding** | Considered as the major activity | Regarded as a small part of the complete development process |
| **Testing** | Not very important, defects are detected much later in the lifecycle | Testing is all encompassing in the development process starting from the requirement specification stage |
| **Documentation** | May or may not exist | Documentation at every stage |

# Modern style of development

- **Key features**
  - Modern software engineering principles are based on error prevention that emphasize detection of errors as close to the point where the errors are committed as possible(phase containment of error).
  - Development of software happens through well defined stages such as requirement specification, design, coding, testing etc.
  - Coding is only a small part of the complete development process
  - Periodic reviews are being carried out during all stages of the development process. The main objective of carrying out reviews is phase containment of errors, i.e. detect and correct errors as soon as possible.

# Modern style of development

- Key features
  - Testing activity has also become all encompassing in the sense that test cases are being developed right from the requirements specification stage.
  - There is better visibility of design and code. By visibility we mean production of good quality, consistent and standard documents during every phase. This has made fault diagnosis and maintenance smoother.
  - Now, projects are first thoroughly planned. Project planning normally includes preparation of various types of estimates, resource scheduling, and development of project tracking plans. Several techniques and tools for tasks such as configuration management, cost estimation, scheduling, etc. are used for effective software project management.
  - Several metrics are being used to help in software project management and software quality assurance.

# Modern style of development

- Other developments in modern s/w development style

✓ Life cycle model

✓ Specification techniques

✓ Project management techniques

✓ Testing techniques

✓ Debugging techniques

✓ Quality assurance techniques

✓ CASE tools(computer aided software engineering tool) etc.

# Software products

Software engineers are concerned with developing software products (i.e., software which can be sold to a customer). There are two kinds of software products:

1. ***Generic products*** These are stand-alone systems that are produced by a development organization and sold on the open market to any customer who is able to buy them. Examples of this type of product include software for PCs such as databases, word processors, drawing packages, and project-management tools. It also includes so-called vertical applications designed for some specific purpose such as library information systems, accounting systems, or systems for maintaining dental records.

# Software products

Software engineers are concerned with developing software products (i.e., software which can be sold to a customer). There are two kinds of software products:

2. ***Customized (or bespoke) products*** These are systems that are commissioned by a particular customer. A software contractor develops the software especially for that customer. Examples of this type of software include control systems for electronic devices, systems written to support a particular business process, and air traffic control systems.

# Categories of software

## System software

System software is a collection of programs written to service other programs. e.g., compilers, editors, and file management utilities, operating system components, drivers, etc.

## Real-time software

Software that monitors/analyses/controls real-world events as they occur is called *real time.*

# Categories of software

## Business software

Business information processing is the largest single software application area. Eg. payroll, accounts receivable/payable, inventory

## Engineering/scientific software

Engineering and scientific software have been characterized by "number crunching" algorithms. Computer aided design tools , system simulation

# Categories of software

## Embedded software

Embedded software resides in read-only memory and is used to control products and systems for the consumer and industrial markets. Embedded software can perform very limited and esoteric functions (e.g., keypad control for a microwave oven.

## Personal Computer Software

Word processing, spread sheets, computer graphics, multimedia, entertainment, database management, personal and business financial applications, external network, and database access are only a few of hundreds of applications. They include all necessary functionality and do not need to be connected to the network.

# Categories of software

## WebApps (Web applications)

The Web pages retrieved by a browser are software that incorporates executable instructions (e.g., CGI,HTML, Perl, or Java), and data (e.g., hypertext and a variety of visual and audio formats).

## Artificial intelligence software

Artificial intelligence (AI) software makes use of nonnumerical algorithms to solve complex problems that are not amenable to computation or straightforward analysis. Expert systems, also called knowledge-based systems, pattern recognition (image and voice), artificial neural networks, theorem proving, and game playing are representative of applications within this category.

# Software Engineering – A Layered Technology

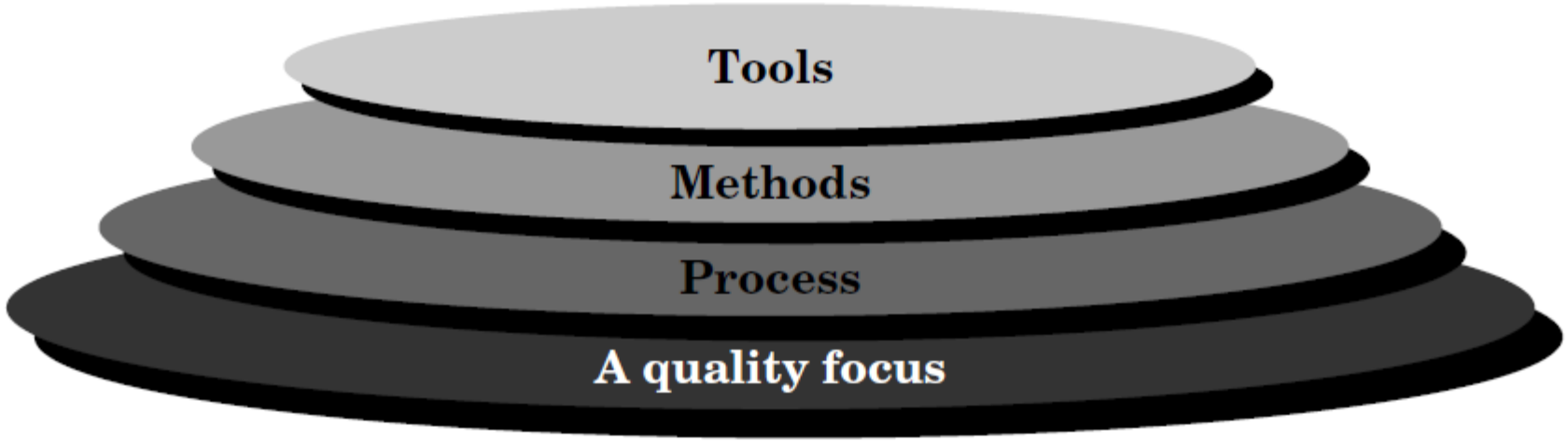
There are different ways in which one can define a software engineering IEEE[IEE93] has developed a comprehensive definition:

Software Engineering is:

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

(2) The study of approaches as in (1).

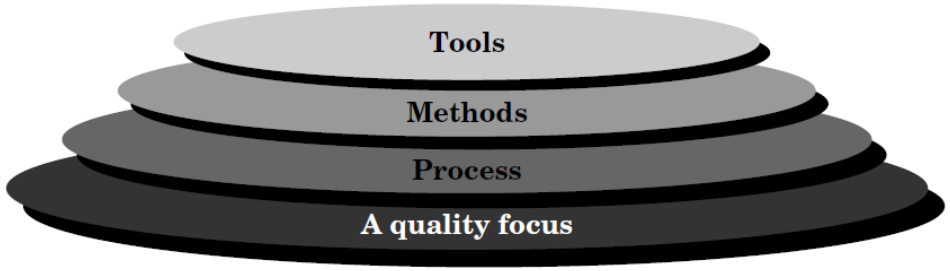
# Software Engineering – A Layered Technology



# Software Engineering – A Layered Technology

## A QUALITY FOCUS

- Any engineering approach (including software engineering) must rest on an organizational commitment to quality.
- Total quality management, six sigma and similar philosophies foster a continuous process improvement culture, and this culture ultimately leads to the development of increasingly more mature approaches to software engineering.
- The base that supports software engineering is a quality focus.



# Software Engineering – A Layered Technology

**PROCESS**

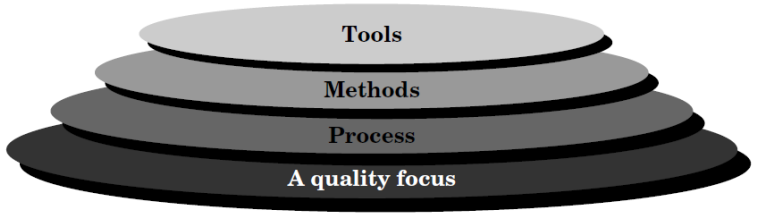
The foundation of software engineering is the process layer.

The complete software engineering process holds the different layers glued together.

It enables rational and timely development of computer software.

Process defines a framework that must be established for effective delivery of software engineering technology.

Basically, a process defines **who is doing what , when and how** to reach a certain goal.

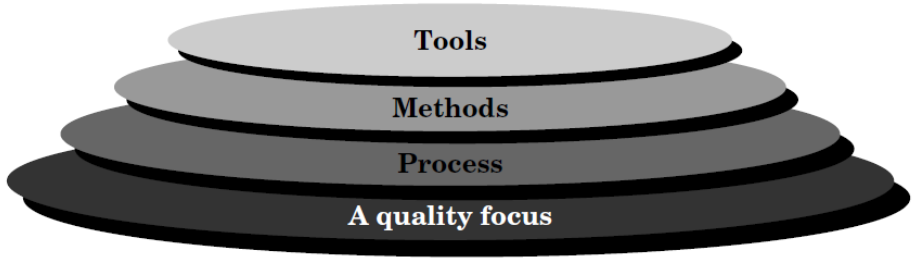


# Software Engineering – A Layered Technology

## METHODS

- Software engineering methods provide the technical how-to's for building software.

- Methods encompass a broad array of tasks that include communication, requirements analysis, design, program construction, testing, and support.

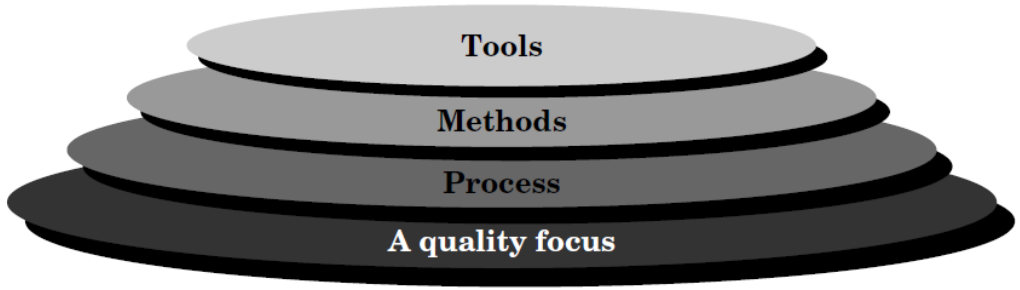


# Software Engineering – A Layered Technology

**TOOLS**

- Tools provide automated or semi automated support for the process and the methods.

- When tools are integrated so that information created by one tool can be used by another, a system for the support of software development, called computer-aided software engineering (CASE), is established.



# Generic View Of Software Engineering

For any entity to be engineered, we must ask the following questions

- What is the problem to be solved?
- What characteristics of the entity are used to solve the problem?
- How will the entity (and the solution) be realized?
- How will the entity be constructed?
- What approach will be used to uncover errors that were made in the design and construction of the entity?
- How will the entity be supported over the long term, when corrections, adaptations, and enhancements are requested by users of the entity.

Entity in question : **“Computer Software”**

# Generic View Of Software Engineering

The work associated with software engineering can be categorized into three generic phases, regardless of application area, project size, or complexity

- Definition phase
- Development phase
- Support phase

# Generic View Of Software Engineering

The **definition phase** focuses on ***what***. That is, during definition, the software engineer attempts to identify

- What information is to be processed?
- What function and performance are desired?
- What system behavior can be expected?
- What interfaces are to be established?
- What design constraints exist?
- What validation criteria are required to define a successful system?

# Generic View Of Software Engineering

The key requirements of the system and the software are identified.

Although the methods applied during the definition phase will vary depending on the software engineering paradigm (or combination of paradigms) that is applied, three major tasks will occur in some form:

- system or information engineering,
- software project planning, and
- requirements analysis.

# Generic View Of Software Engineering

The development phase focuses on *how*. That is, during development a software engineer attempts to define

- how data are to be structured?
- how function is to be implemented within a software architecture?
- how procedural details are to be implemented?
- how interfaces are to be characterized?
- how the design will be translated into a programming language (or nonprocedural language)?
- how testing will be performed?

The methods applied during the development phase will vary, but three specific technical tasks should always occur: software design, code generation, and software testing.

# Generic View Of Software Engineering

The **support phase** focuses on ***change*** associated with error correction, adaptations required as the software's environment evolves, and changes due to enhancements brought about by changing customer requirements.

The support phase reapplies the steps of the definition and development phases but does so in the context of existing software.

# Generic View Of Software Engineering

Four types of change are encountered during the support phase:

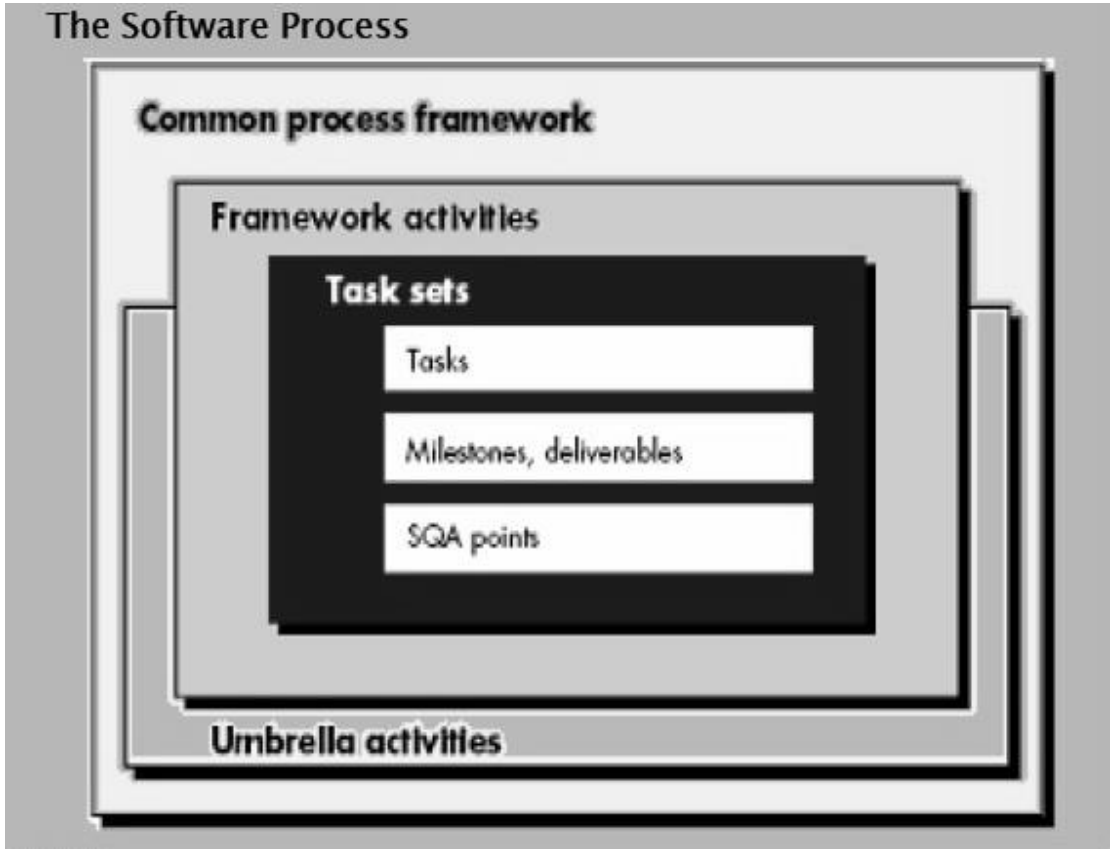
- **Correction.** *Corrective maintenance* changes the software to correct defects.
- **Adaptation.** *Adaptive maintenance* results in modification to the software to accommodate changes to its external environment.
- **Enhancement.** *Perfective maintenance* extends the software beyond its original functional requirements.
- **Prevention.** Computer software deteriorates due to change, and because of this, *preventive maintenance*, often called *software reengineering*, must be conducted to enable the software to serve the needs of its end users. In essence, preventive maintenance makes changes to computer programs so that they can be more easily corrected, adapted, and enhanced.

# Generic View Of Software Engineering

The phases and related steps described in our generic view of software engineering are complemented by a number of umbrella activities which are applied throughout the software process. Typical activities in this category include:

- Software project tracking and control
- Formal technical reviews
- Software quality assurance
- Software configuration management
- Document preparation and production
- Reusability management
- Measurement
- Risk management

# Software Process



# Software Process

## A common process framework

A process framework establishes the foundation for a complete software process.

How..?? By identifying a small number of framework activities that are applicable to all software projects, regardless of their size or complexity.

## Task sets

A number of task sets—each a collection of software engineering work tasks, project milestones, work products, and quality assurance points—enable the framework activities to be adapted to the characteristics of the software project and the requirements of the project team.

# Software Process

## Umbrella Activities

Finally, umbrella activities—such as software quality assurance, software configuration management, and measurement—overlay the process model.

**Umbrella activities are independent of any one framework activity and occur throughout the process.**

# Capability Maturity Model

In recent years, there has been a significant emphasis on "process maturity." The

Software Engineering Institute (SEI) has developed a comprehensive model predicated on a set of software engineering capabilities that should be present as organizations reach different levels of process maturity.

To determine an organization's current state of process maturity, the SEI uses an assessment that results in a five point grading scheme.

The grading scheme determines compliance with a *capability maturity model* (CMM) that defines key activities required at different levels of process maturity.

# Capability Maturity Model

The SEI approach provides a measure of the global effectiveness of a company's software engineering practices and establishes five process maturity levels that are defined in the following manner:

Level 1 : Initial

Level 2 : Repeatable

Level 3 : Defined

Level 4 : Managed

Level 5 : Optimizing

# Capability Maturity Model

## Level 1 : Initial

The software process is characterized as ad hoc and occasionally even chaotic. Few processes are defined, and success depends on individual effort.

## Level 2 : Repeatable

Basic project management processes are established to track cost, schedule, and functionality. The necessary process discipline is in place to repeat earlier successes on projects with similar applications.

# Capability Maturity Model

## Level 3 : Defined

The software process for both management and engineering activities is documented, standardized, and integrated into an organization-wide software process. All projects use a documented and approved version of the organization's process for developing and supporting software. This level includes all characteristics defined for level 2.

# Capability Maturity Model

## Level 4 : Managed

Detailed measures of the software process and product quality are collected. Both the software process and products are quantitatively understood and controlled using detailed measures. This level includes all characteristics defined for level 3.

## Level 5 : Optimizing

Continuous process improvement is enabled by quantitative feedback from the process and from testing innovative ideas and technologies. This level includes all characteristics defined for level 4.

# Capability Maturity Model

The five levels defined by the SEI were derived as a consequence of evaluating responses to the SEI assessment questionnaire that is based on the CMM. The results of the questionnaire are distilled to a single numerical grade that provides an indication of an organization's process maturity.

The SEI has associated key process areas (KPAs) with each of the maturity levels.

The KPAs describe those software engineering functions (e.g., software project planning, requirements management) that must be present to satisfy good practice at a particular level.

# Capability Maturity Model

Each KPA is described by identifying the following characteristics:

***Goals***—the overall objectives that the KPA must achieve.

***Commitments***—requirements (imposed on the organization) that must be met to achieve the goals or provide proof of intent to comply with the goals.

***Abilities***—those things that must be in place (organizationally and technically) to enable the organization to meet the commitments.

***Activities***—the specific tasks required to achieve the KPA function.

***Methods for monitoring implementation***—the manner in which the activities are monitored as they are put into place.

***Methods for verifying implementation***—the manner in which proper practice for the KPA can be verified.

# Capability Maturity Model

Eighteen KPAs (each described using these characteristics) are defined across the maturity model and mapped into different levels of process maturity. The following KPAs should be achieved at each process maturity level:

## Process maturity level 2

**Software configuration management**

**Software quality assurance**

**Software subcontract management**

**Software project tracking and oversight**

**Software project planning**

**Requirements management**

# Capability Maturity Model

Process maturity level 3

Peer reviews

Intergroup coordination

Software product engineering

Integrated software management

Training program

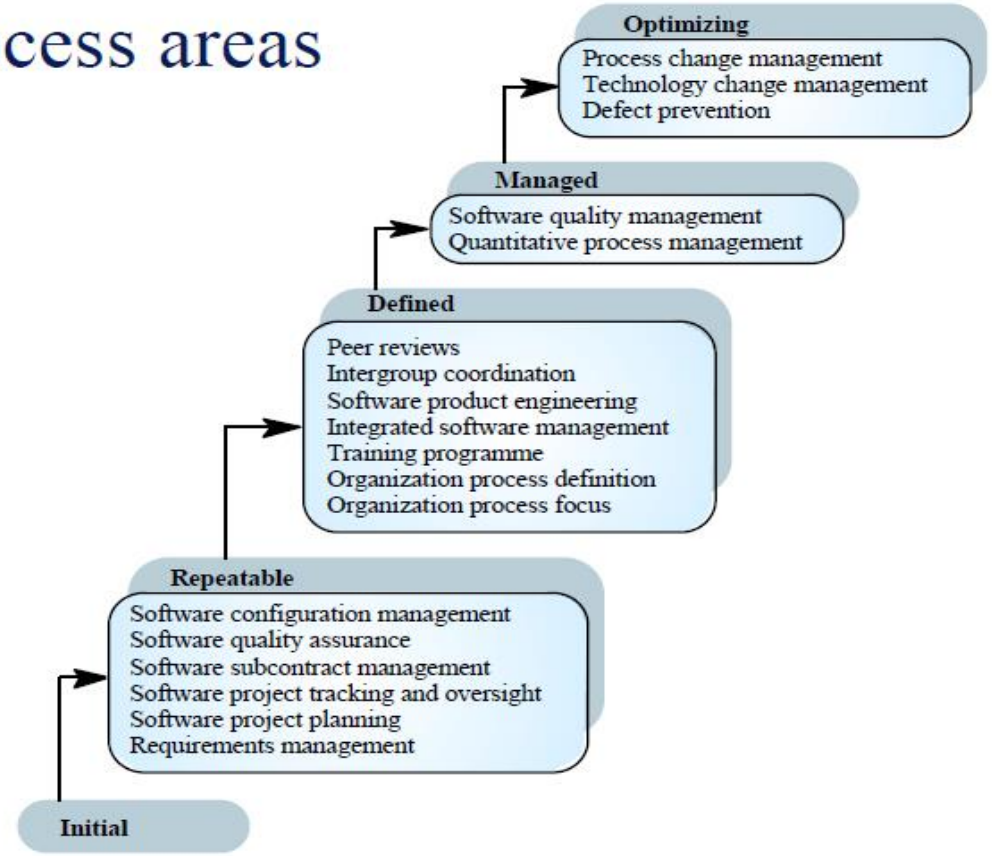
Organization process definition

Organization process focus

- Process maturity level 4
- Software quality management
- Quantitative process management

- Process maturity level 5
- Process change management
- Technology change management
- Defect prevention

# Capability Maturity Model



# Capability Maturity Model

Each of the KPAs is defined by a set of *key practices* that contribute to satisfying its goals.

The key practices are policies, procedures, and activities that must occur before a key process area has been fully instituted.

The SEI defines *key indicators* as "those key practices or components of key practices that offer the greatest insight into whether the goals of a key process area have been achieved."

Assessment questions are designed to probe for the existence (or lack thereof) of a key indicator.