# Learning Resource
# On
# Introduction to Software Engineering

## Chapter-1
## Introduction to Software Engineering

**Prepared By:**
**Kunal Anand, Asst. Professor**
**SoCE, KIIT, DU, Bhubaneswar-24**

**Source:** Fundamentals of Software Engineering, Rajib Mall, 5th Edn., PHI

# Chapter Outcomes:

- Define Software Engineering
- Explain the scope and need of Software Engineering
- Role of Software Engineer
- Identify software components
- Describe Software Crisis
- Distingusih between Programs and Software Products
- Identify characteristics of good software
- Discuss evolution of Software Engineering
- Differentiate between Exploratory style and Modern Software Engineering practices.

# Organization of this Chapter:

- What is Software Engineering?
- Scope and Necessity of Software Engineering
- Need of Software Engineering
- Role of Software Engineer
- Software components
- Software Crisis
- Programs vs. Software Products
- Characteristics of good software
- Evolution of Software Engineering
- Difference between Exploratory style and Modern Software Engineering practices.

# What is Software Engineering?

- “A systematic collection of good program development practices and techniques.”

- Software Engineering discusses systematic and cost-effective techniques for software development.

- Systematic collection of past experiences:
  - Techniques
  - Methodologies
  - Guidelines

# Software + Engineering

- The term software engineering is composed of two words, **software** and **engineering**.
- **Software** is a collection of executable programming code, associated libraries and documentations. Software, when made for a specific requirement is called software product.
- **Engineering** on the other hand, is all about developing products, using well-defined, scientific principles and methods.
- So, we can define software engineering as an engineering discipline associated with the development of software product using well-defined scientific principles, methods and procedures.
- ***The outcome of software engineering is an efficient and reliable software product.***

# Scope and Necessity of Software Engineering

- Ex: Difference between building a Wall and a Multi-storied Building.

- Without using software engineering principles, it would be difficult to develop large programs.

- In industry, it is usually needed to develop large programs to accommodate multiple functions.

- A problem with developing such large commercial programs is that the complexity and difficulty levels of the programs increase exponentially with their sizes.

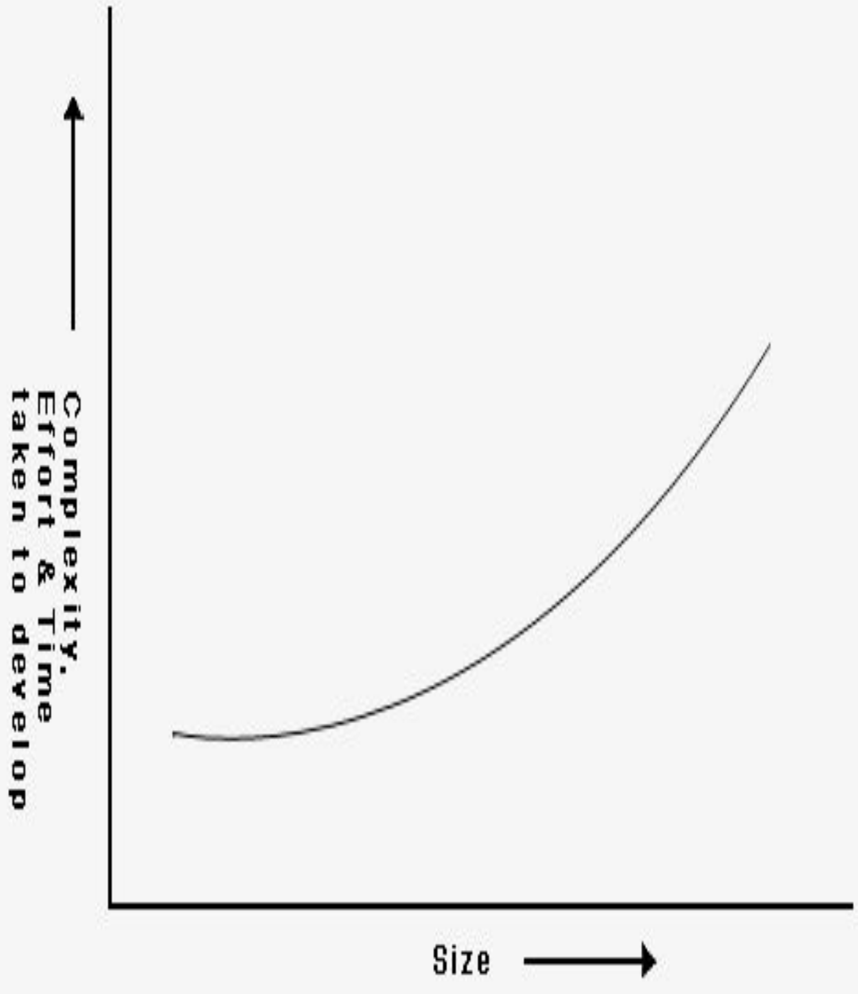


**Fig. 1.1:** Increase in development time and effort with problem size

- For example, a program of size 1,000 lines of code has some complexity.

- But a program with 10,000 LOC is not just 10 times more difficult to develop but may as well turn out to be 100 times more difficult unless software engineering principles are used.

- In such situations software engineering techniques come to rescue. Software engineering helps to reduce the programming complexity.

- Software engineering principles use two important techniques to reduce problem complexity: **Abstraction** and **Decomposition**.

- **Abstraction:** It implies that a problem can be simplified by omitting irrelevant details.

- Consider only those **aspects of the problem that are relevant for certain purpose** and suppress other aspects that are not relevant.

- Once the simpler problem is solved, then the omitted details can be taken into consideration to solve the next lower-level abstraction, and so on.

- Abstraction is a powerful way of reducing the complexity of the problem.

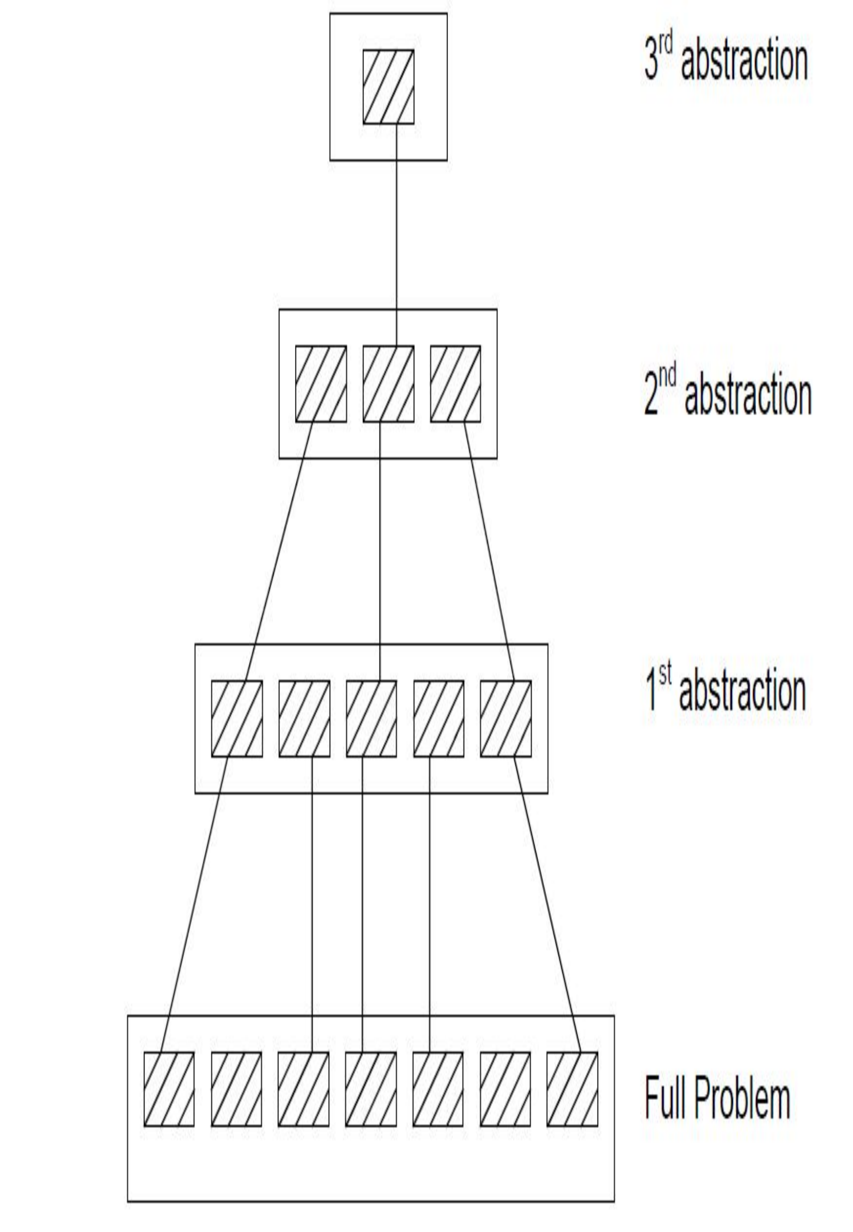


**Fig. 1.2:** Abstraction

- The other approach to tackle problem complexity is **Decomposition**. A complex problem is divided into several smaller problems and then the smaller problems are solved one by one. However, in this technique any random decomposition of a problem into smaller parts will not help.
- The problem must be decomposed such that each component of the **decomposed problem can be solved independently** and then the solution of the different components can be combined to get the full solution.
- **Challenge:** A good decomposition of a problem as should minimize interactions among various components.
  - If the different subcomponents are interrelated, then the different components cannot be solved separately and the desired reduction in complexity will not be realized.

# Decomposition

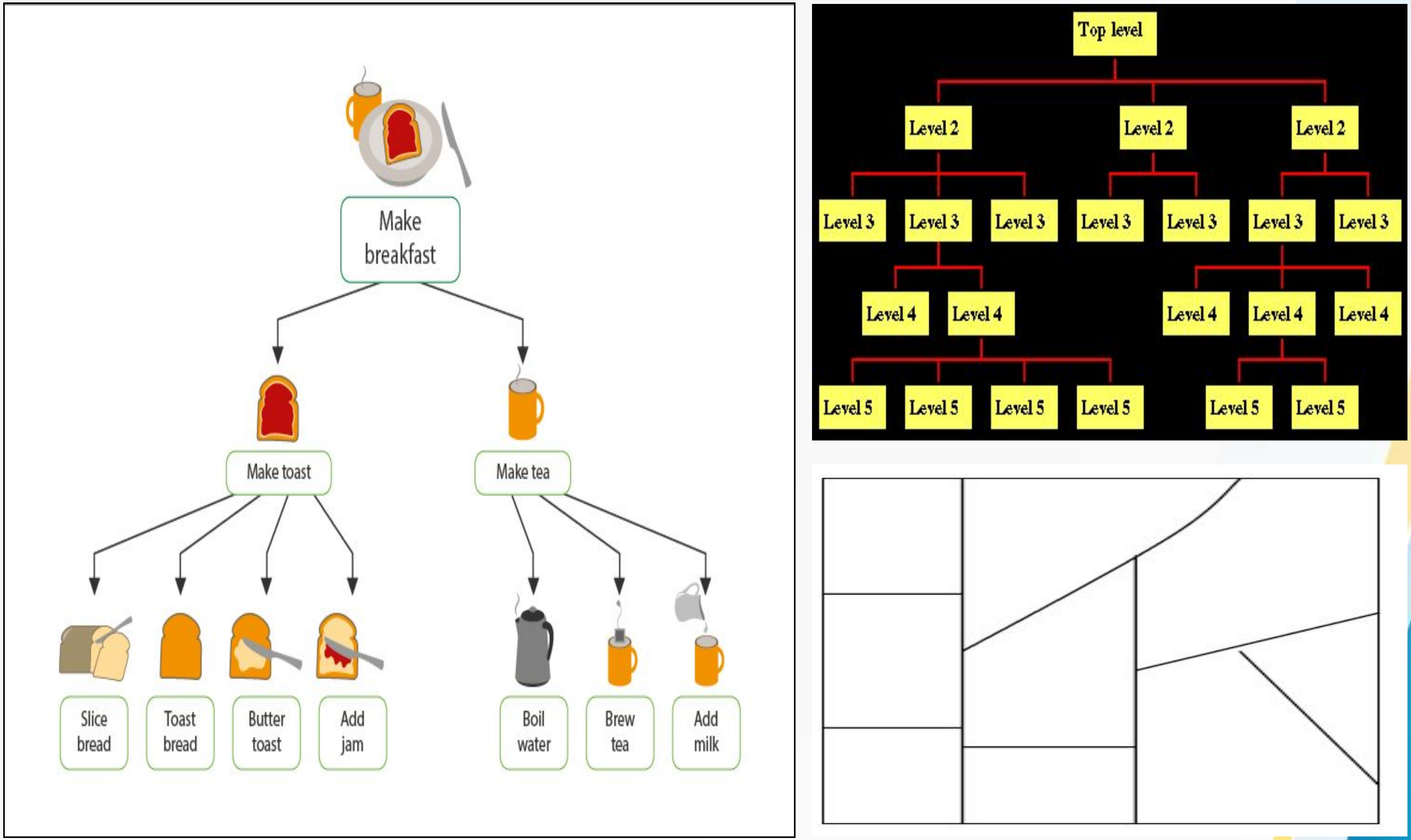


**Fig. 1.3:** Decomposition

# Technology Development Pattern

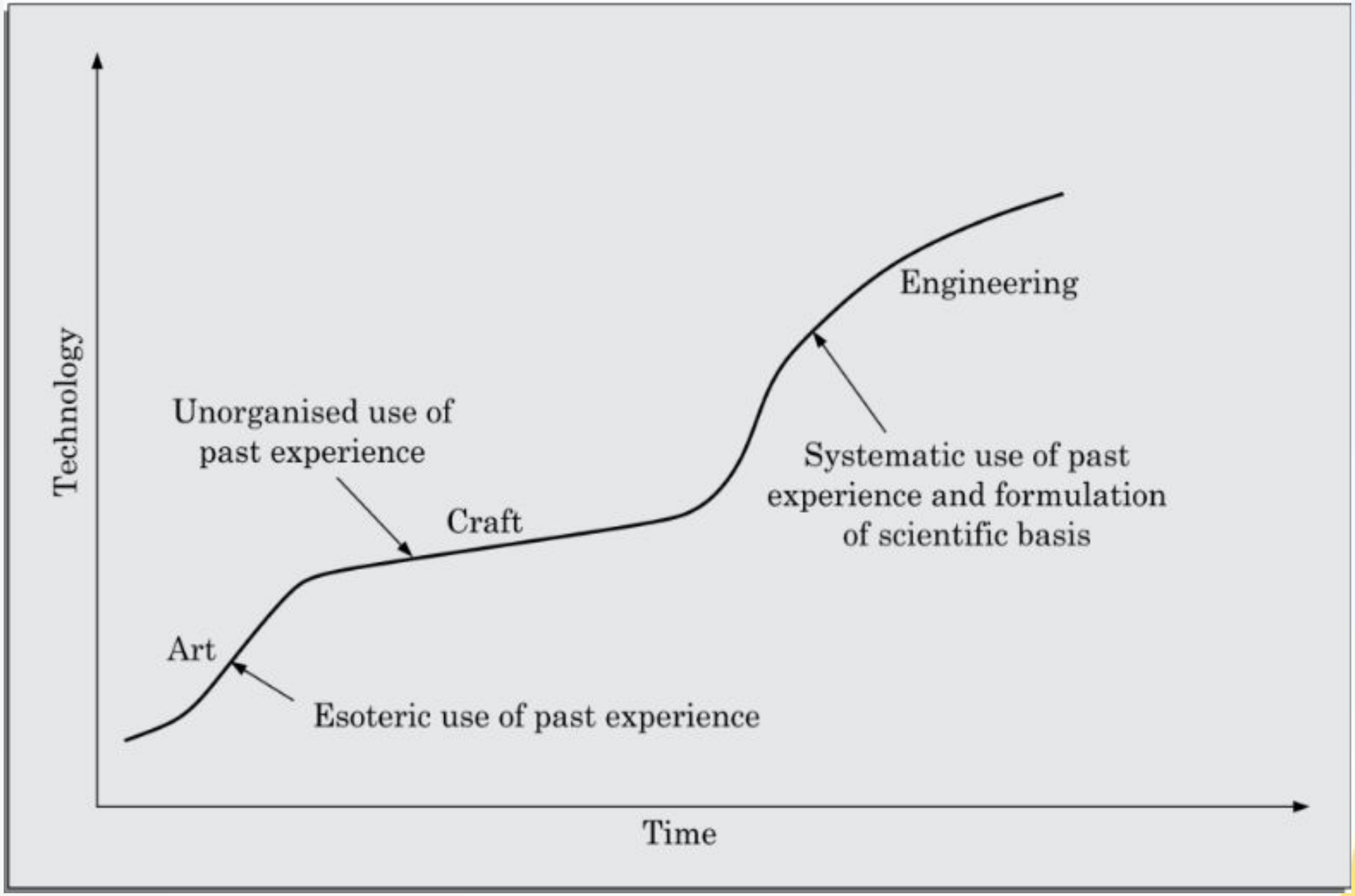


**Figure 1.4 :** Evolution of technology with time

***Esoteric** : intended for or likely to be understood by only a small number of people with a specialized knowledge or interest.*

# Need of Software Engineering

- **Large software** - It is easier to build a wall than to a house or building, likewise, as the size of software become large engineering must step in to give it a scientific process.

- **Scalability-** If the software process were not based on scientific and engineering concepts, it would be easier to re-create new software than to scale an existing one.

- **Cost-** As hardware industry has shown its skills and huge manufacturing has lower down the price of computer and electronic hardware. But the cost of software remains high if proper process is not adapted.

# Need of Software Engineering

- **Dynamic Nature-** The always growing and adapting nature of software hugely depends upon the environment in which the user works.
  - If the nature of software is always changing, new enhancements need to be done in the existing one. This is where software engineering plays a good role.

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

# Role of Software Engineer

Mentioned below are the key responsibilities that software engineers have to take upon themselves:

- Analysis
- Development
  - Design
  - Code
  - Test
- Deployment
- Maintenance
- Innovation
- Quality assurance
- Planning and project management

# Software Components

- Software components are the building blocks of software applications.
- In SE, **component-based development** has become a cornerstone for creating robust and scalable systems.
- Component-based design involves breaking down complex systems into manageable and reusable units, commonly referred to as components.
- These components exhibit specific functionality and they communicate via well-defined interface.
- The components must have the characteristics like reusability, independence, interoperability, scalability, encapsulation.
- Some common examples of software components are Application Programming Interface (API), Libraries, Framework, Modules, Plugins, Widget, Controls and Connectors.

# Software Crisis

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

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

# Software Crisis (cont.)

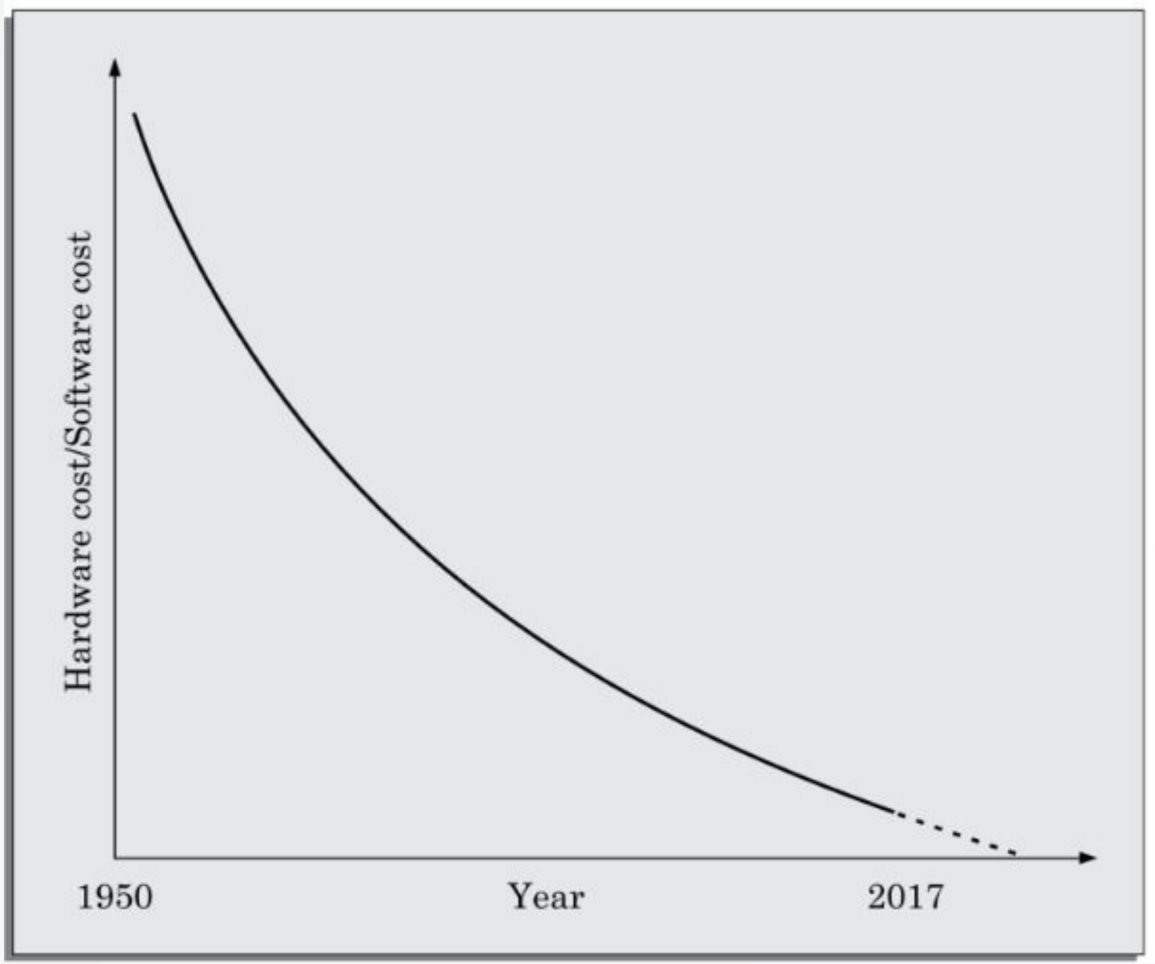


**Fig. 1.5:** Relative changes of hardware and software costs over time

*At present, organizations are spending much more on software than hardware.*

# Programs versus Software Products

- Usually small in size
- Author himself is sole user
- Single developer
- Lacks proper user interface
- Lacks proper documentation
- Ad hoc development.

- Large in size
- Large number of users
- Team of developers
- Well-designed interface
- Well documented & user-manual prepared
- Systematic development

# Characteristics of Good Software

- A software product can be judged by what it offers and how well it can be used.
- This software must satisfy on the following grounds:
  - **Operational:** This tells us how well software works in operations. It can be measured on:
    - Budget : in terms of cost, manpower
    - Usability : ease of use
    - Efficiency : minimum expenditure of time and effort
    - Correctness: with respect to a specification
    - Functionality : having a practical use
    - Dependability : extent to which a critical system is trusted by its users
    - Security : degree of resistance or protection from

# Transitional

- This aspect is important when the software is moved from one platform to another:
  - Portability : usability of the same software in different environments
  - Interoperability : ability of a system or a product to work with other systems
  - Reusability : use of existing assets in some form within the software product development process
  - Adaptability : able to change or be changed in order to fit or work better in some situation

# Maintenance

- This aspect briefs about how well a software has the capabilities to maintain itself in the ever-changing environment:
  - Modularity: degree to which a system's components may be separated and recombined
  - Maintainability: degree to which an application is understood, repaired, or enhanced
  - Flexibility: ability for the solution to adapt to possible or future changes in its requirements
  - Scalability: ability of a program to scale

# Evolution of software design techniques

- 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. 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.**

- 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.

- 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.

- In late 1960s  to cope with this problem, particular attention to the design of the program's **control flow structure** was advised which introduced **"GOTO"** statement.
- The use of "**GOTO**" statements in high-level languages were very natural because of their familiarity with **JUMP** statements which are very frequently used in assembly language programming.
  - But sometimes GOTO makes the loop very complex and considered harmful.
- Due to the challenges of previous structure, 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.**

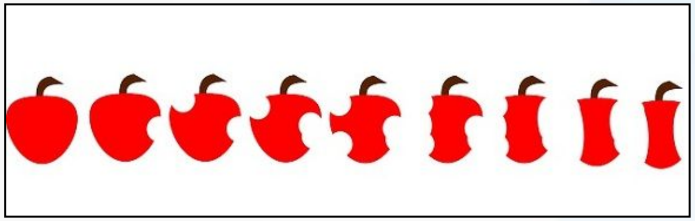
# Control Flow-Based Design (Late 60s)

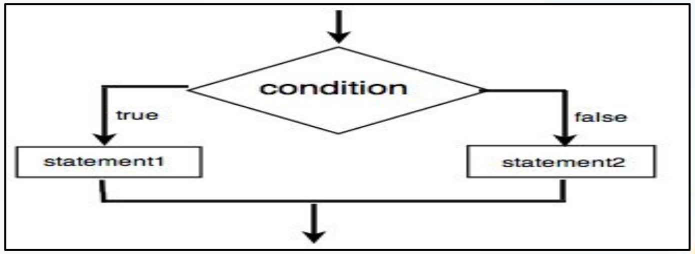
## Structured program

- 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;)





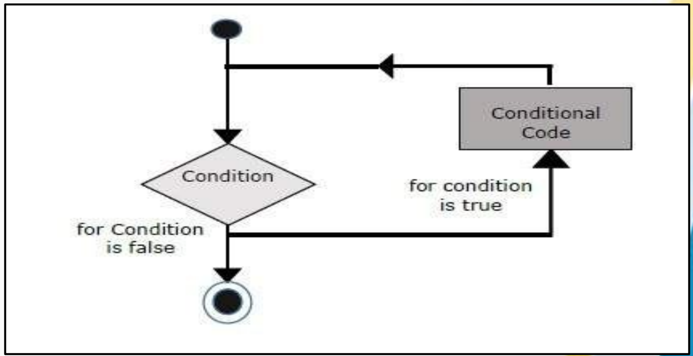


**Fig. 1.6:** Programming constructs

- After structured programming, the next important development was **data structure-oriented design.**

- 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.

- Example of a very popular data structure-oriented design technique is **Jackson's Structured Programming (JSP)** methodology, developed by Michael Jackson in the1970s.

- 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.

- **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.

# Differences between the Exploratory style and Modern software development practices

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

- Emphasis has shifted from **error correction to error prevention.** Modern practices emphasize detection of errors as close to their point of introduction as possible.

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

# contd..

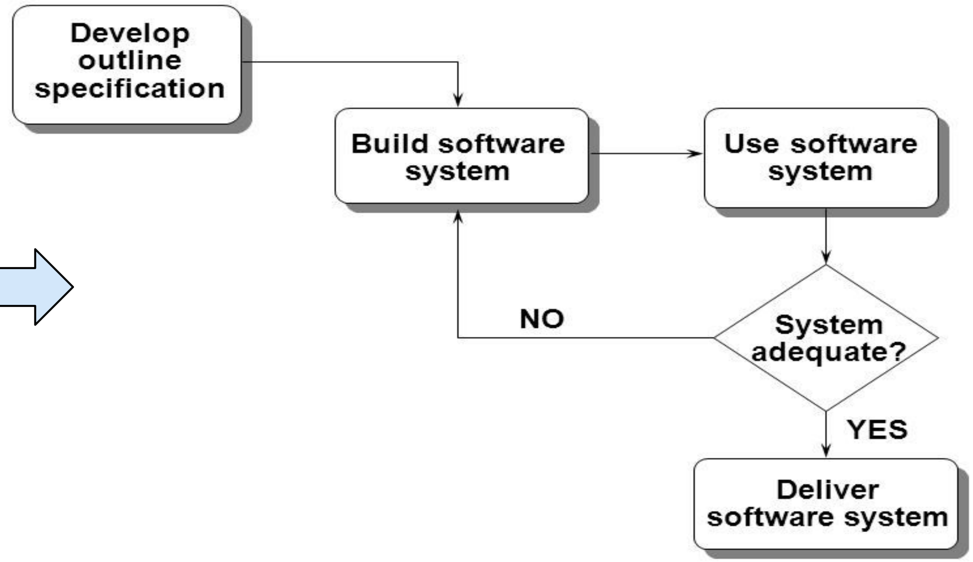
- In **exploratory style**, coding is synonymous with program development. However, in **modern SE practices**, coding is considered only a small part of program development effort.
- A lot of effort and attention is now being paid to **requirements specification**. Also, now there is a distinct **design phase** where standard design techniques are being used. This was not the case in **exploratory programming**.
- In **modern SE practices**, periodic reviews are being carried out during all stages of development process. **Exploratory programming** did not pay any attention towards any review.
- Now, Software testing has become systematic as standard testing techniques are available. This is was not in the case of exploratory programming.

# contd..

- In the past, very little attention was being given to producing **good quality and consistent documents**. Modern SE practices puts lot of effort in producing good quality documentation for the software under development.
  - Because of good documentation, fault diagnosis and maintenance are smoother now. Several metrics are being used to help in software project management, quality assurance, etc.

- **Exploratory style** of programming was heavily dependent on individual heroics for developing good quality software. However, **modern SE practices** encourages the use of Computer-aided Software Engineering (CASE) tools.

**Exploratory Programming**



**Modern Software development process**

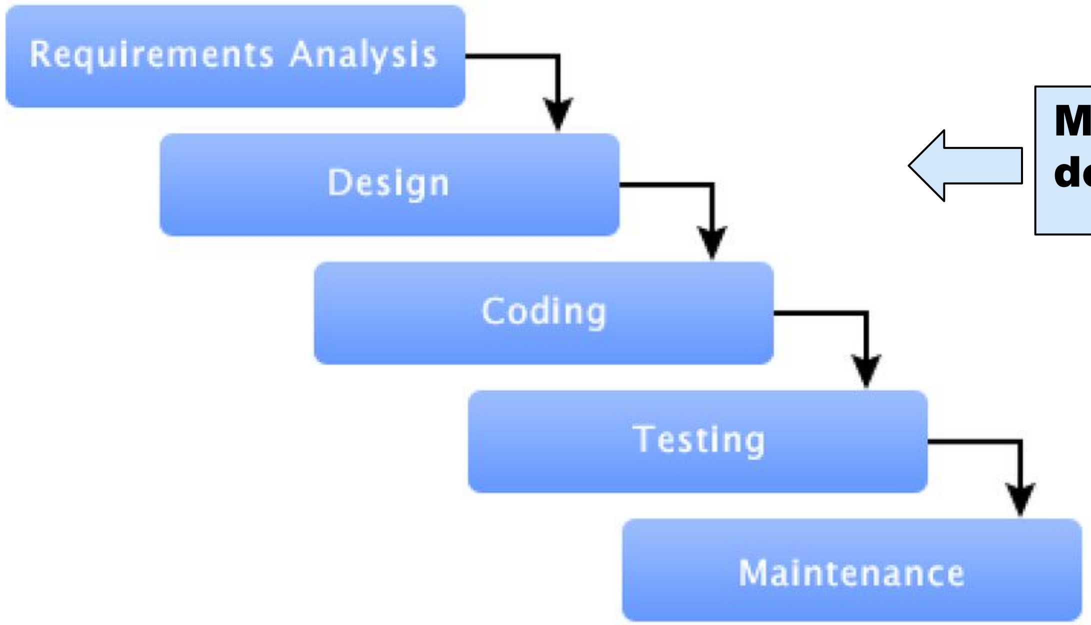


**Fig. 1.7:** Exploratory programming vs Modern SE practices