Foundations of Computer Science
Software World

ONE LOVE. ONE FUTURE.

HUST

hust.edu.vn fb.com/dhbkhn

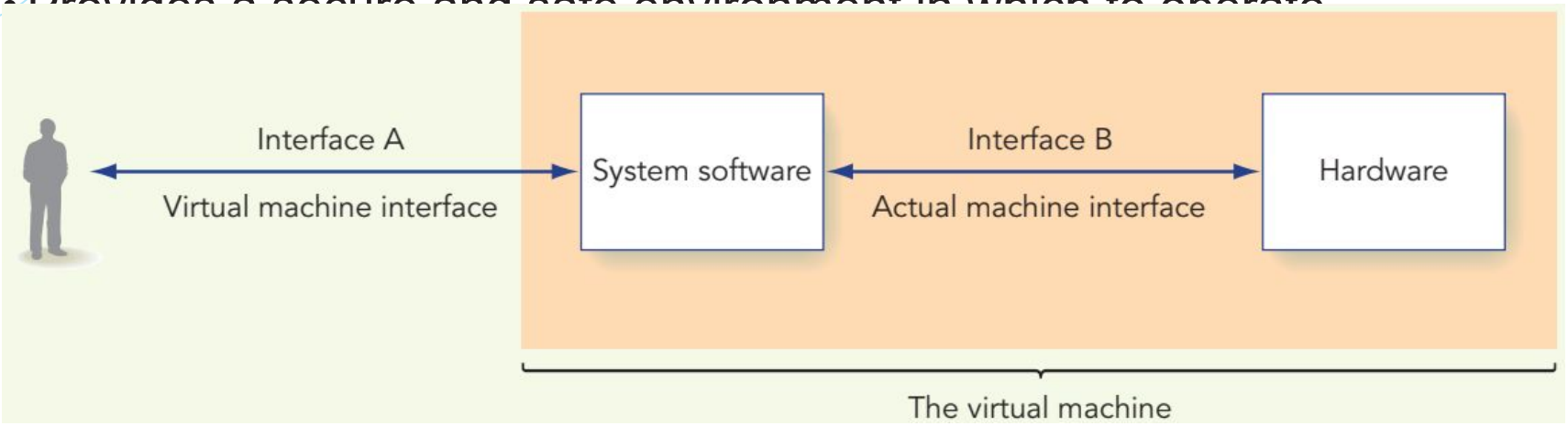
FaMI 1956

Contents

- System Software
- Assembler
- Assembly Language
- Translation and Loading
- Operation System
- High Level Programming Language
- Procedural Programming Language
- Compiler
- Software Development Life Cycle
- Development Environment
- Development Models

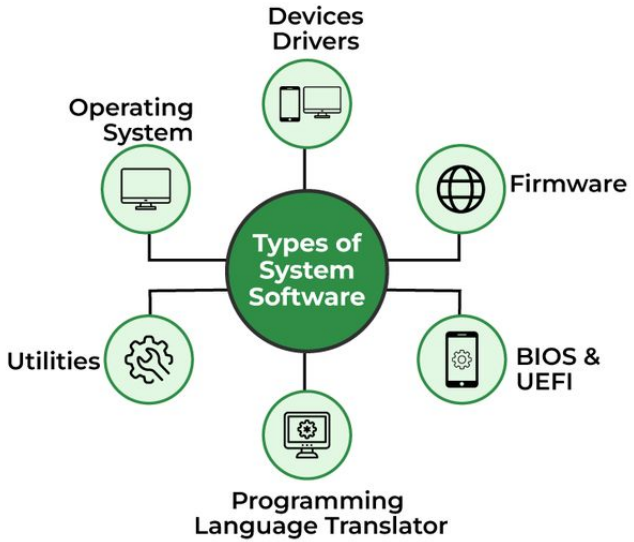
System Software as an Interface

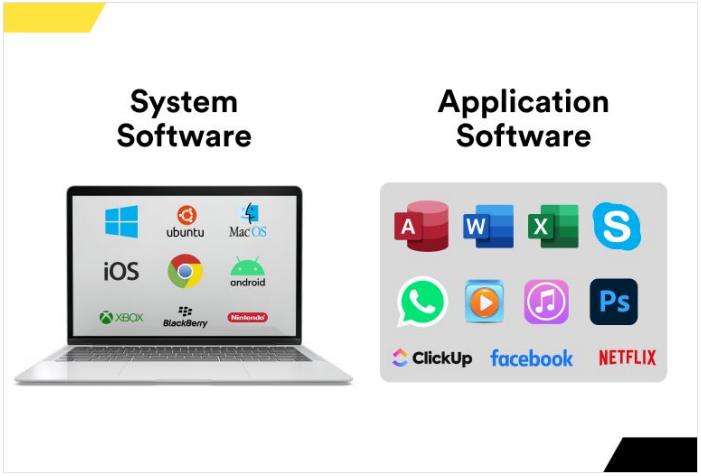
- Hides from the user the messy details of the underlying hardware
- Hides the complex and unimportant (to the user) details of the internal structure of the Von Neumann architecture
- Presents information about what is happening in a way that does not require in-depth knowledge of the internal structure of the system
- Allows easy user access to the resources available on this computer
- Prevents accidental or intentional damage to hardware, programs, and data
- Provides a secure and safe environment in which to operate



System Software

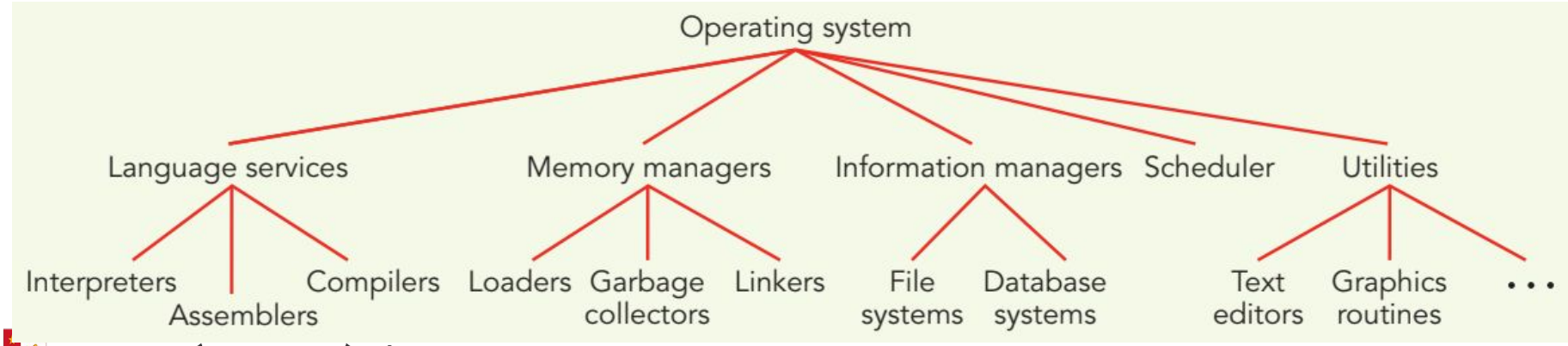
- **System software** is a collection of computer programs that manage the resources of a computer and facilitates access to those resources.
- This contrasts with application software that allows a user to address some specialized task of interest to that user, for example, write a document, create an image, browse the web, or solve a system of equations.





Types of System Softwares

- *User interface*:
  - **graphical user interface (GUI)** giving intuitive visual overview as well as graphical control of the capabilities and services of the computer
  - controlled by keystrokes, mouse clicks, finger taps, voice activation, or biometric scans such as fingerprints.
- *Language services*:
  - called *assemblers*, *compilers*, and *interpreters*
  - allow writing programs in a high-level, user-oriented language rather than the machine language
  - often include components such as text editors and debuggers.



Types of System Softwares

- *Memory managers*:
  - allocate memory space for programs and data
  - retrieve space when no longer needed.
- *Information managers:*
  - handle the organization, storage, and retrieval of information on storage devices such as the hard drives, DVDs, flash drives, and tapes, remotely in data centers (*cloud storage*).
  - organize information in an efficient hierarchical manner, using directories, folders, and files
  - keep data safe from accidental or intentional misuse.
- *I/O* systems: allow easy and efficient using the many different types of input and output devices

Types of System Softwares

- *Scheduler*
  - keeps a list of programs ready to run on the processor
  - selects the one that will execute next
  - allows having several different programs active at a single time, e.g. surfing the web while waiting finish printing
- *Utilities*:
  - collections of library routines
  - provide a wide range of useful services to a user or to other system routines.
  - e.g. text editors, online help routines, image and sound applications, and control panels
  - sometimes called **program libraries**.

Types of System Softwares

- *Scheduler*
  - keeps a list of programs ready to run on the processor
  - selects the one that will execute next
  - allows having several different programs active at a single time, e.g. surfing the web while waiting finish printing
- *Utilities*:
  - collections of library routines
  - provide a wide range of useful services to a user or to other system routines.
  - e.g. text editors, online help routines, image and sound applications, and control panels
  - sometimes called **program libraries**.

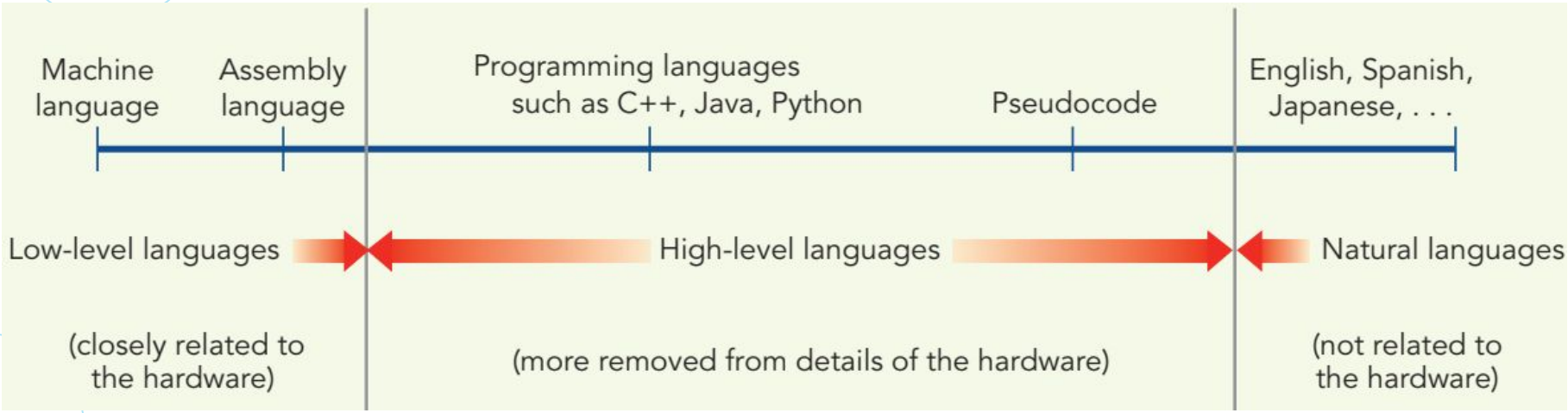
Assembly vs. Machine Language

-

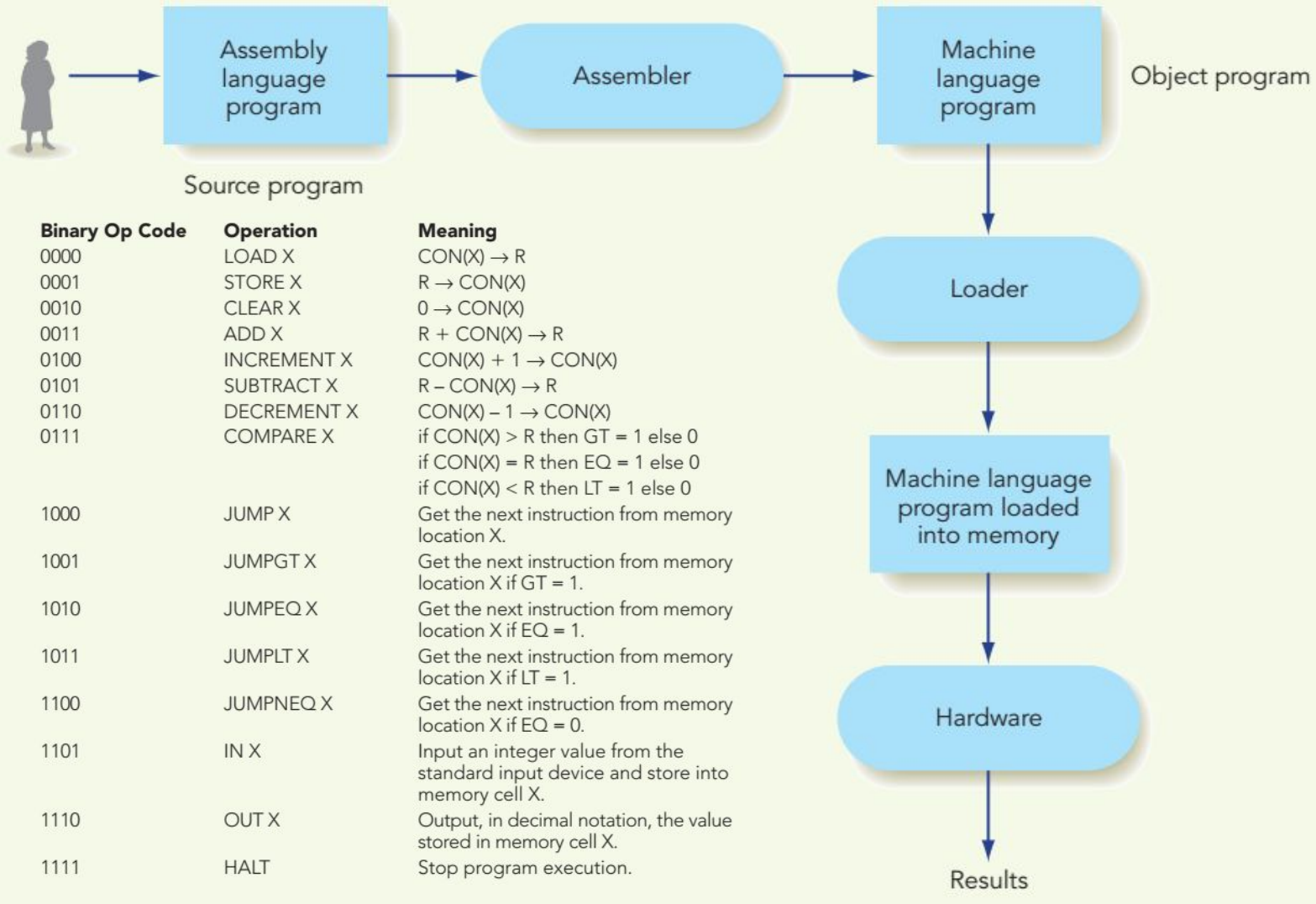
Continuum of Programming Languages

- Assembly languages:
  - **low-level programming languages**
  - Second generation programming languages
  - Each symbolic assembly language instruction is translated into exactly *one* binary machine language instruction
- Assembly vs. Machine Language
  - Use of symbolic operation codes rather than numeric (binary) ones
  - Use of symbolic memory addresses rather than numeric (binary) ones
  - Pseudo-operations providing useful user-oriented services such as data generation
- **High-level programming languages**:
  - Pascal, C++, Java, Python, …
  - User oriented, not machine specific
  - Use both natural language and mathematical notation
  - A single high-level language instruction is typically translated into *many* machine language instructions
  - Virtual environment created by a high-level language is far more powerful

Continuum of Programming Languages



Translation, Loading, and Execution Processes



Binary Op Code	Operation	Meaning
0000	LOAD X	CON(X) → R
0001	STORE X	R → CON(X)
0010	CLEAR X	0 → CON(X)
0011	ADD X	R + CON(X) → R
0100	INCREMENT X	CON(X) + 1 → CON(X)
0101	SUBTRACT X	R – CON(X) → R
0110	DECREMENT X	CON(X) – 1 → CON(X)
0111	COMPARE X	if CON(X) > R then GT = 1 else 0 if CON(X) = R then EQ = 1 else 0 if CON(X) < R then LT = 1 else 0
1000	JUMP X	Get the next instruction from memory location X.
1001	JUMPGT X	Get the next instruction from memory location X if GT = 1.
1010	JUMPEQ X	Get the next instruction from memory location X if EQ = 1.
1011	JUMPLT X	Get the next instruction from memory location X if LT = 1.
1100	JUMPNEQ X	Get the next instruction from memory location X if EQ = 0.
1101	IN X	Input an integer value from the standard input device and store into memory cell X.
1110	OUT X	Output, in decimal notation, the value stored in memory cell X.
1111	HALT	Stop program execution.

Assembly Examples

```
        LOAD    ONE     --Put a 1 into register R
        STORE   I       --Store the constant 1 into I
        ⋮
        INCREMENT I     --Add 1 to the contents of memory location I
        ⋮               --The following .DATA statements should be
                          placed after the HALT instruction
I:      .DATA   0       --The memory location holding the variable I,
                          initially set to 0
ONE:    .DATA   1       --The integer constant 1
```

```
        LOAD    B       --Put the value B into register R
        ADD     C       --R now holds the sum (B + C)
        SUBTRACT SEVEN  --R now holds (B + C – 7)
        STORE   A       --Store the result into A
        ⋮               --The following .DATA statements should
                          be placed after the HALT instruction
A:      .DATA   0
B:      .DATA   0
C:      .DATA   0
SEVEN:  .DATA   7       --The integer constant 7
```

Assembly Examples

```
        IN        X       --Read the first data value
        IN        Y       --and now the second
        LOAD      X       --Load the value of X into register R
        COMPARE   Y       --Compare Y to X and set the condition codes
                            according to the outcome of the comparison
        JUMPGT    DONE    --If Y is greater than X, go to DONE
        OUT       X       --We get here only if X ≥ Y, so print X
        JUMP      NEXT    --Skip over the next instruction and continue
DONE:   OUT       Y       --We get here if Y > X, so print Y
        ⋮
NEXT:   ⋮                 --The program continues here

                          --The following .DATA statements go after
                            the HALT instruction
X:      .DATA     0       --Space for the two data values
Y:      .DATA     0
        ⋮
```

Assembly Examples

Step	*Operation*
1	Set *i* to 0
2	While the value of *i* < 10,000 do
3–8	⋮
9	Add 1 to the value of *i*
10	End of the loop
11	Stop

```
            LOAD    ZERO        --Initialize the loop counter to 0
            STORE   I           --This is Step 1 of the algorithm
LOOP:       LOAD    MAXVALUE    --Put the integer value 10,000
                                  into register R
            COMPARE I           --Compare I against 10,000
            JUMPEQ  DONE        --If I = 10,000 we are done (Step 2)
            ⋮                   --Here is the loop body (Steps 3–8)
            INCREMENT I         --Add 1 to I (Step 9)
            JUMP    LOOP        --End of the loop body (Step 10)
DONE:       HALT                --Stop execution (Step 11)
ZERO:       .DATA   0           --This is the constant 0
I:          .DATA   0           --The loop counter; it goes from
                                --0 to 10,000
MAXVALUE:   .DATA   10000       --Maximum number of executions
            ⋮
```

Assembly Examples

Step	Operation
1	Set the value of *Sum* to 0
2	Input the first number *N*
3	While *N* is not negative do
4	Add the value of *N* to *Sum*
5	Input the next data value *N*
6	End of the loop
7	Print out *Sum*
8	Stop

```
            .BEGIN                  --This marks the start of the program
            CLEAR       SUM         --Set the running sum to 0 (line 1)
            IN          N           --Input the first number N (line 2)
--The next three instructions test whether N is a negative number (line 3)
AGAIN:      LOAD        ZERO        --Put 0 into register R
            COMPARE     N           --Compare N and 0
            JUMPLT      NEG         --Go to NEG if N < 0
--We get here if N ≥ 0. We add N to the running sum (line 4)
            LOAD        SUM         --Put SUM into R
            ADD         N           --Add N. R now holds (N + SUM)
            STORE       SUM         --Put the result back into SUM
--Get the next input value (line 5)
            IN          N
--Now go back and repeat the loop (line 6)
            JUMP        AGAIN
--We get to this section of the program only when we encounter a negative
value
NEG:        OUT         SUM         --Print the sum (line 7)
            HALT                    --and stop (line 8)
--Here are the data generation pseudo-ops
SUM:        .DATA       0           --The running sum goes here
N:          .DATA       0           --The input data are placed here
ZERO:       .DATA       0           --The constant 0
--Now we mark the end of the entire program
            .END
```

Assembler

1. Convert symbolic op codes to binary.
2. Convert symbolic addresses to binary.
3. Perform the assembler services requested by the pseudo-ops.
4. Put the translated instructions into a file for future use.

Operation	Binary Value
ADD	0011
CLEAR	0010
COMPARE	0111
DECREMENT	0110
HALT	1111
OUT	1110
⋮	
STORE	0001
SUBTRACT	0101

Op Code Table

Binary Search

Label	Code		Location Counter
LOOP:	IN	X	0
	IN	Y	1
	LOAD	X	2
	COMPARE	Y	3
	JUMPGT	DONE	4
	OUT	X	5
	JUMP	LOOP	6
DONE:	OUT	Y	7
	HALT		8
X:	.DATA	0	9
Y:	.DATA	0	10

(a)

Symbol	Address Value
LOOP	0
DONE	7
X	9
Y	10

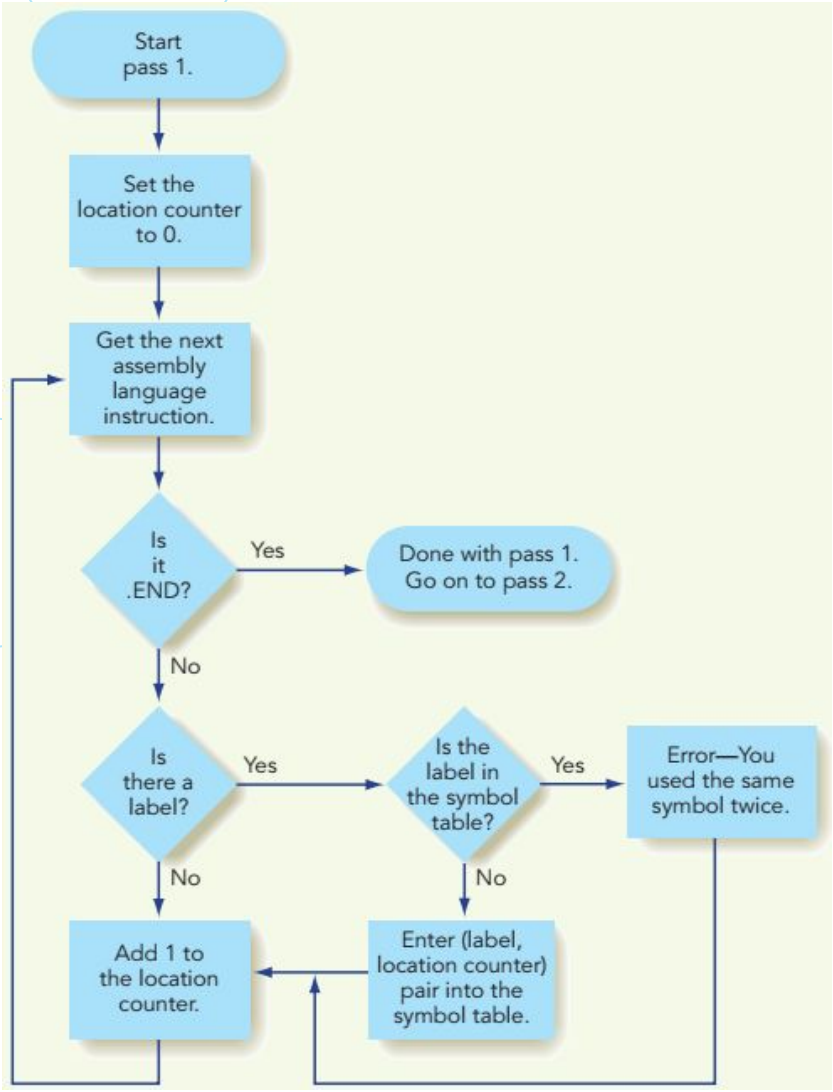
(b)

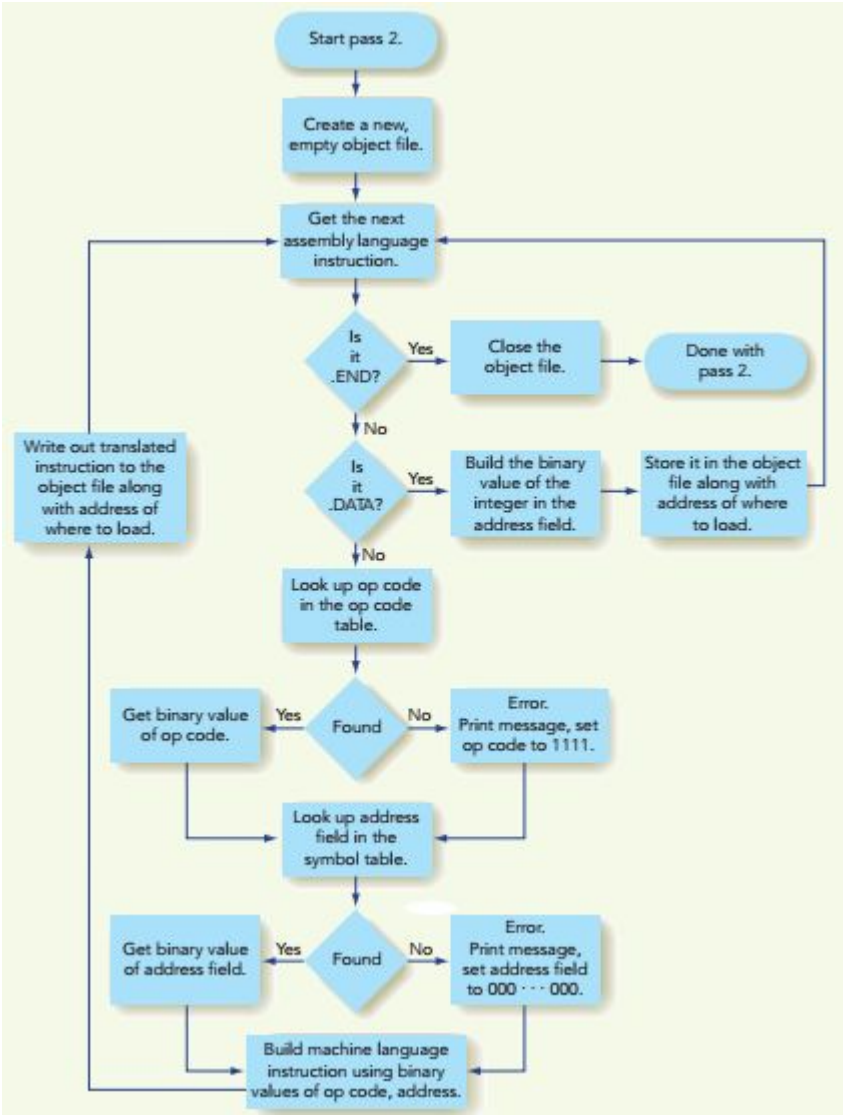
Symbol Table Generation

Pass: the process of examining and processing every assembly language instruction in the program, one instruction at a time.

Binding: associate symbolic name with a physical memory address

Translation





Object Program Example

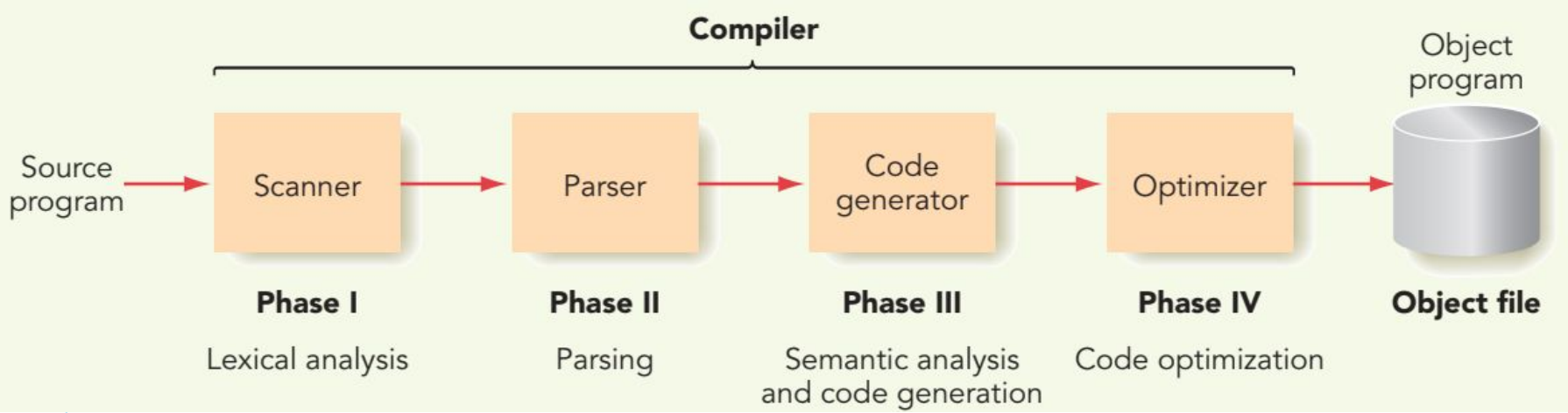
Instruction Format:

Op Code	Address
4 bits	12 bits

Object Program:

Address	*Machine Language Instruction*	*Meaning*
0000	1101 000000001001	IN X
0001	1101 000000001010	IN Y
0010	0000 000000001001	LOAD X
0011	0111 000000001010	COMPARE Y
0100	1001 000000000111	JUMPGT DONE
0101	1110 000000001001	OUT X
0110	1000 000000000000	JUMP LOOP
0111	1110 000000001010	OUT Y
1000	1111 000000000000	HALT
1001	0000 000000000000	The constant 0
1010	0000 000000000000	The constant 0

Compiler Structure

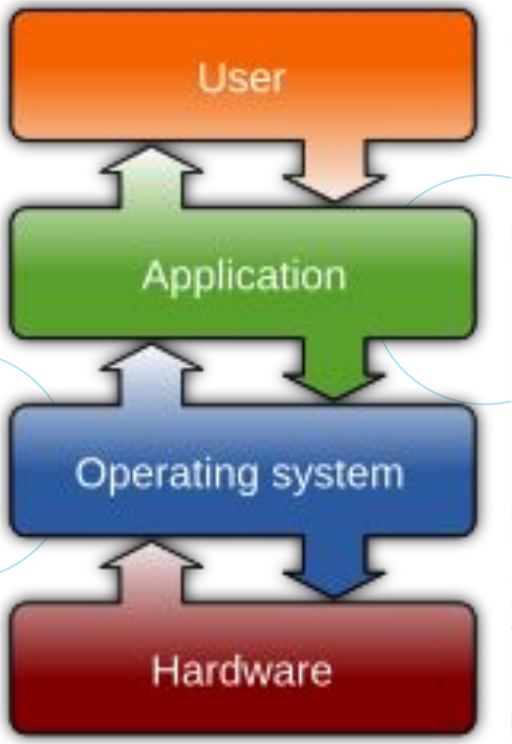


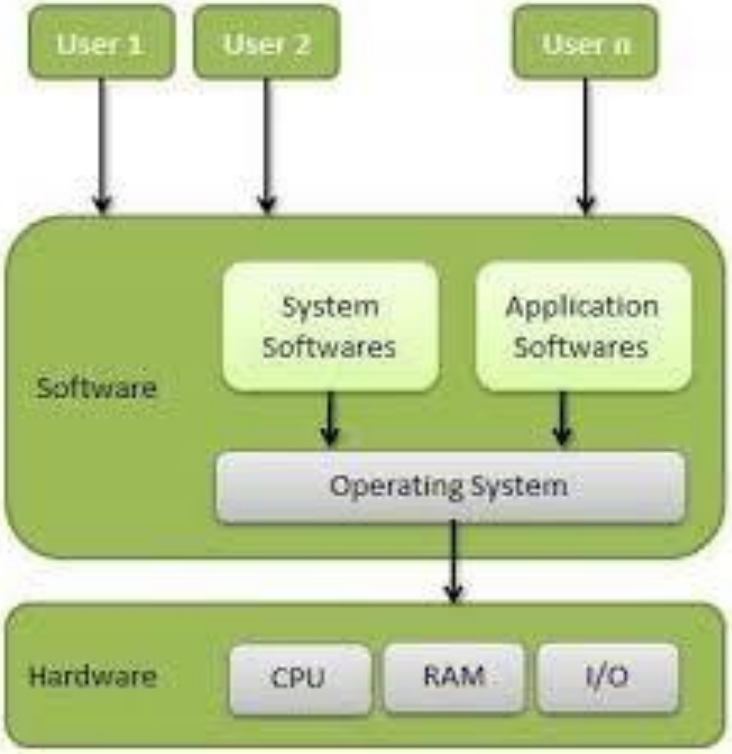
- *Lexical analysis*—The compiler examines the individual characters in the source program and groups them into syntactical units, called tokens
- *Parsing*—The sequence of tokens is checked if it is syntactically correct according to the rules of the programming language

Compiler Structure

- *Semantic analysis and code generation*—the compiler analyzes
statements' meaning and generates the proper sequence of machine language instructions to carry out the intended actions
- *Code optimization*—The compiler takes the generated code and sees whether it can be made more efficient, either by making it run faster, having it occupy less memory, or possibly both

Operating System





- An **operating system** (**OS**) is system software that manages computer hardware and software resources, and provides common services for computer programs.

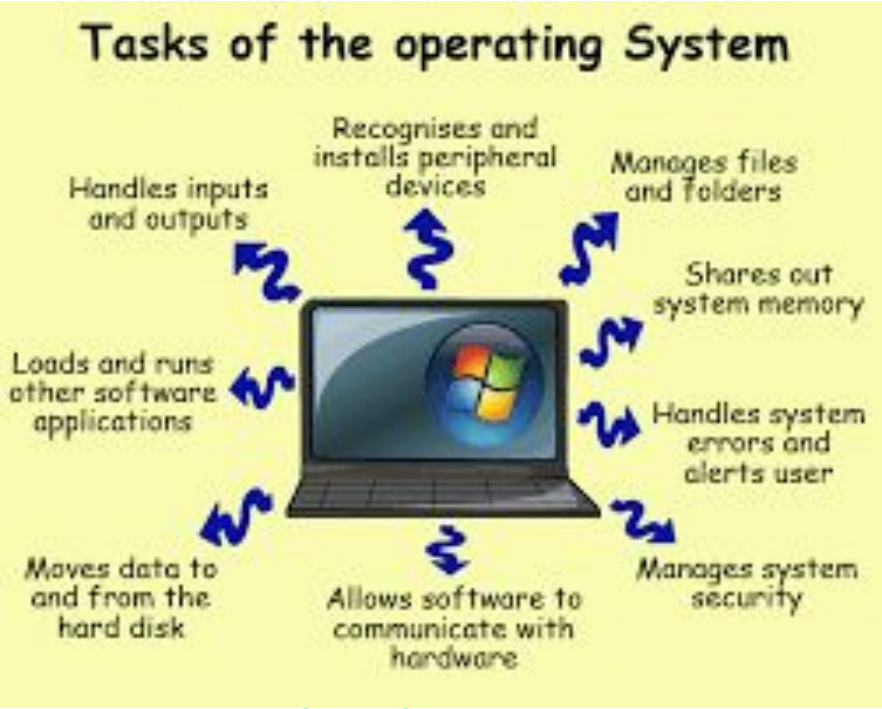
Operating Systems Functions

- User interface management (a receptionist)
- Control of access to the system and to data files (a security guard)
- Program scheduling and activation (a dispatcher)
- Efficient resource allocation (an efficiency expert)
- Deadlock detection and error detection (a traffic officer)

File: GRADES

Name	Permitted Operations
Smith	R (R = Read only)
Jones	RA (A = Append)
Adams	RAC (C = Change)
Doe	RACD (D = Delete)

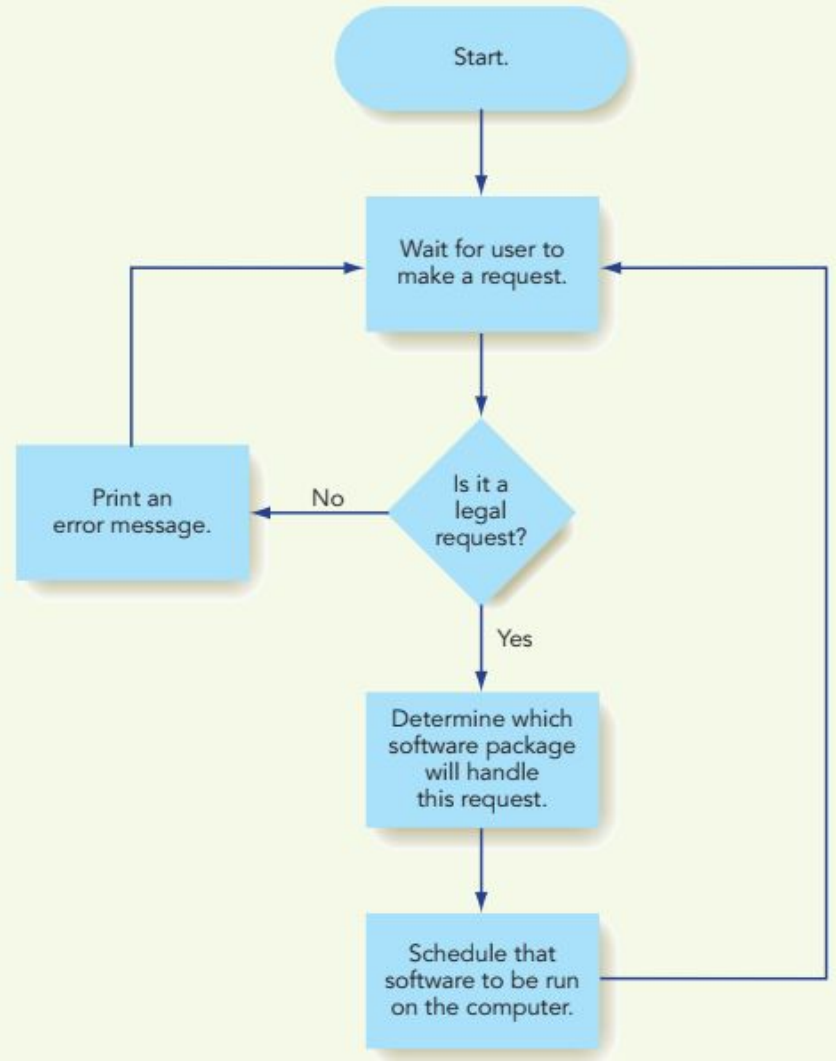
Operating Systems Components



Operating Systems Command Examples

- Load a program into memory
- Run a program
- Save information in a file or a directory
- Retrieve a file previously stored on the local machine or in the cloud
- List all the files for the user
- Delete or rename a file
- Display a file on a specified device
- Copy a file from one device to another
- Establish a network connection
- Let the user set or change one of the current machine settings
- Tell how much memory or data storage is currently in use
- Put the device to sleep or turn it off

User Interface Response



Timeline of Operating Systems

Timeline of Operating Systems

1956 — GM-NAA I/O, produced by General Motors for its IBM 704

1977 — Apple][released

1981 — MS-DOS is released by Microsoft

1991 — Linux is released by Linus Torvalds

1995 — Windows 95 is released

2008 — Android is released (based on a Linux kernel)

2011 — OpenShift released by Red Hat

1960s — IBM develops a series of OSs for its 360 series. Multics is developed and abandoned but UNIX is developed as a consequence.

1970s — Unix becomes popular in academic circles and spawns many versions

1980s — The home computer revolution

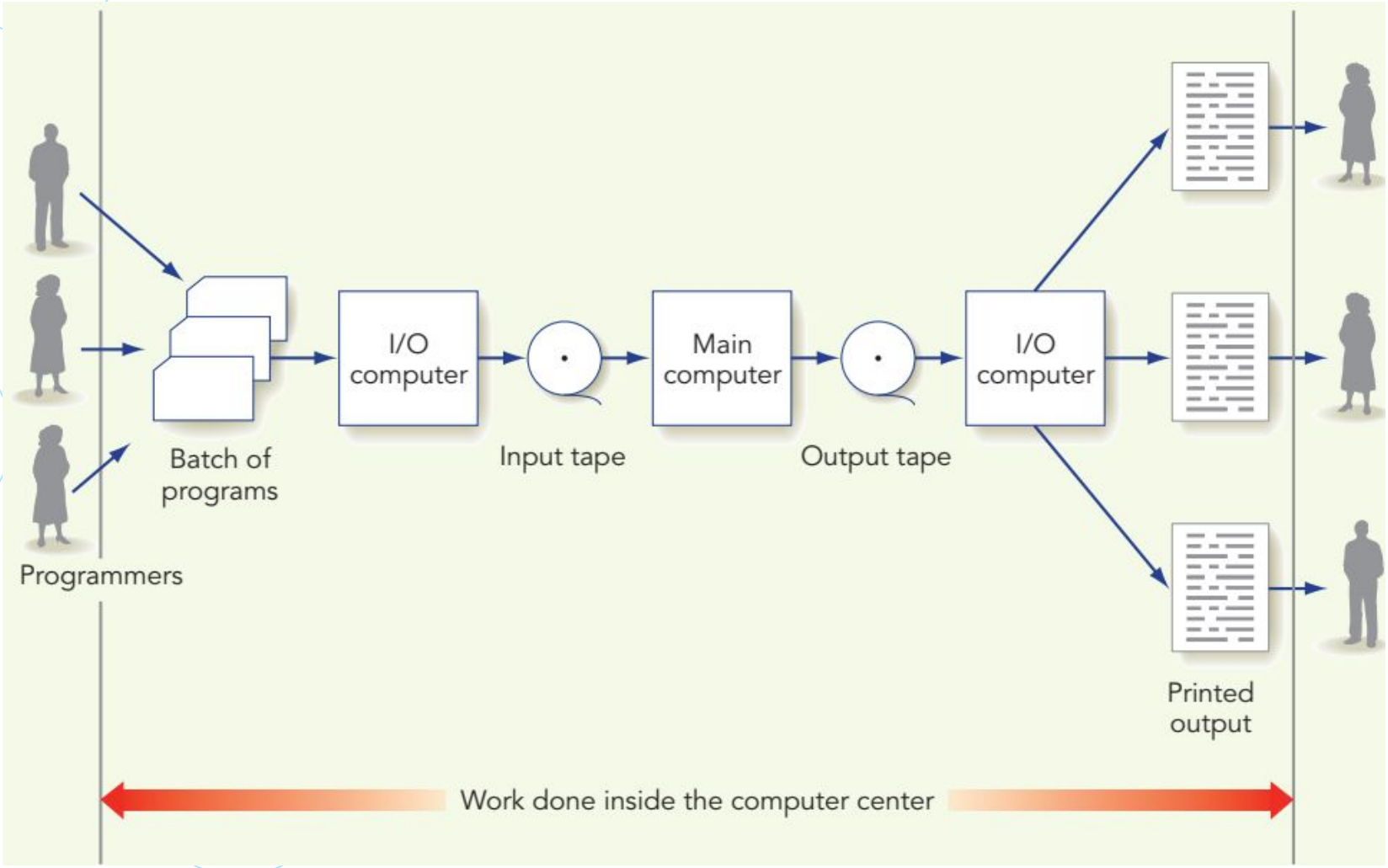
1990s — Windows dominates the laptop and desktop market

2000s — Unix and then Linux dominate the Supercomputer Market

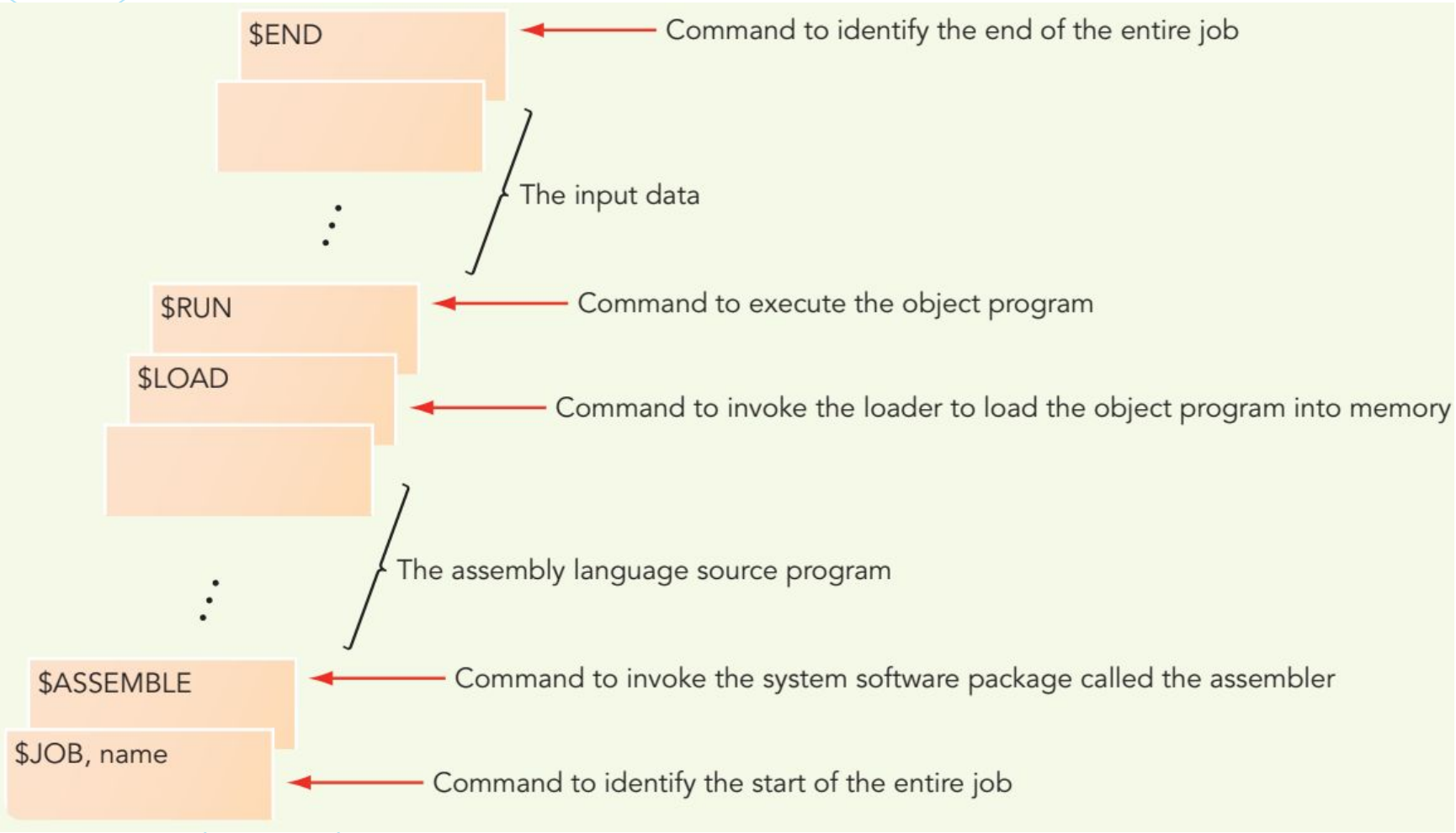
2010s — Smart phones become ubiquitous after the iPhone release in 2007

Types of Operating System

Batch Computer System (2nd Generation)

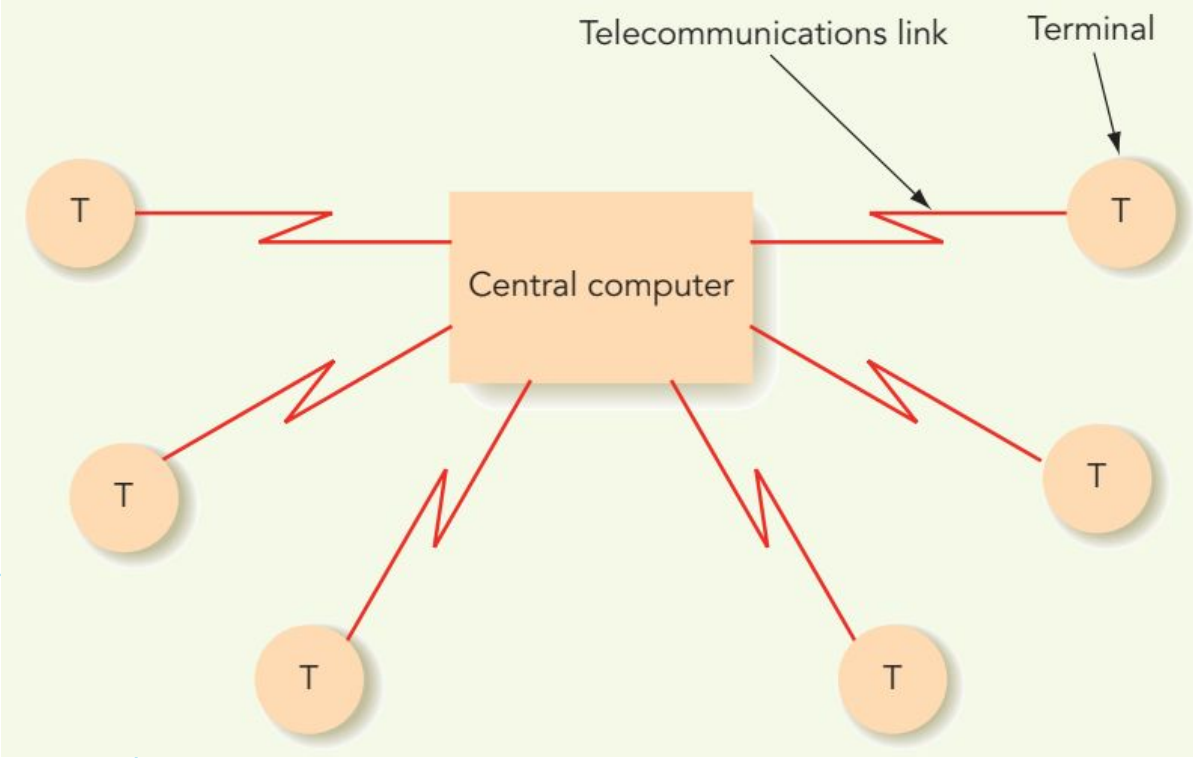


A Batch Job Example

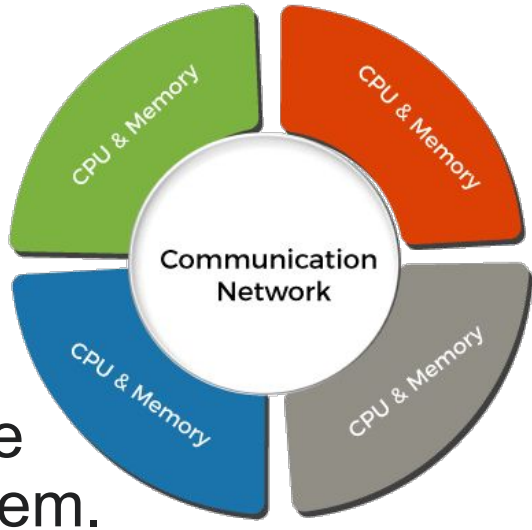


Time-sharing Operating System (3rd Generation Operating System)

- Time-sharing operating systems schedule tasks for efficient use of the system and may also include accounting software for cost allocation of processor time, mass storage, peripherals, and other resources.

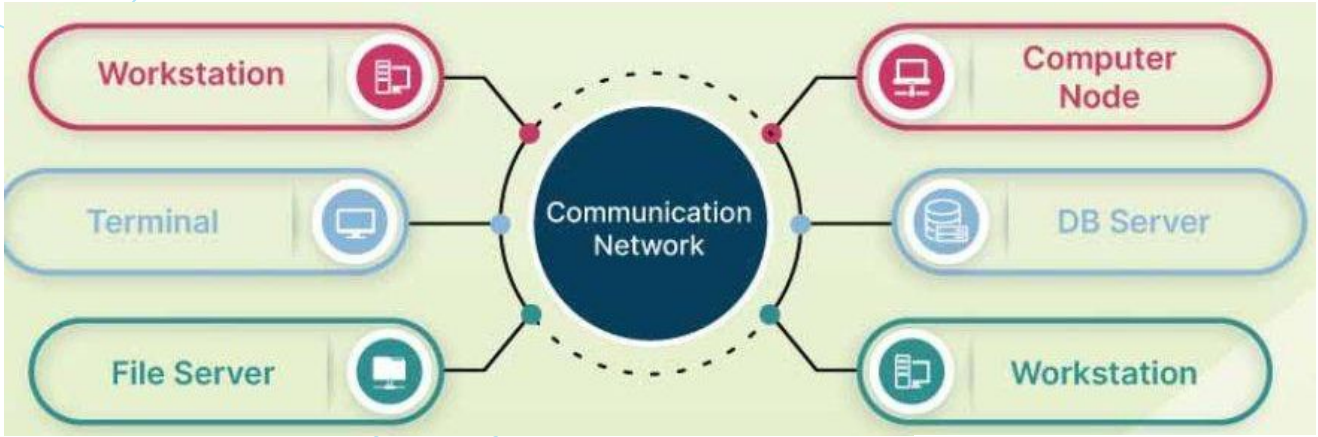


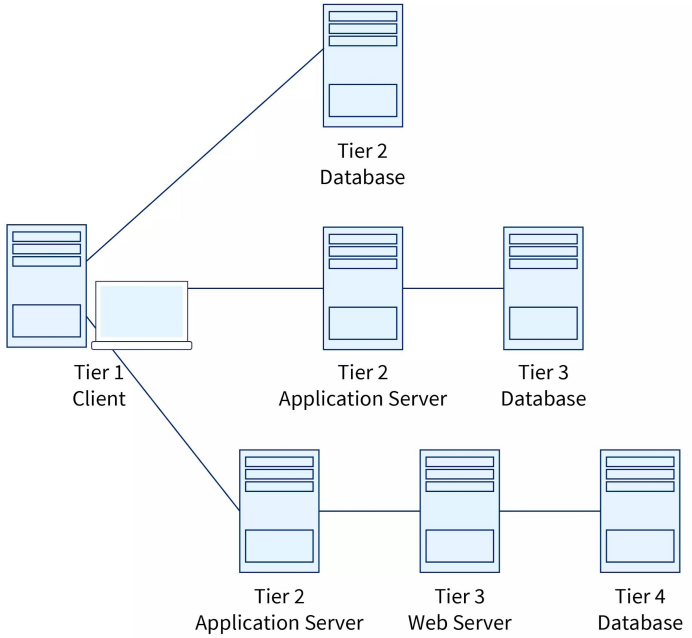
Distributed Operating System



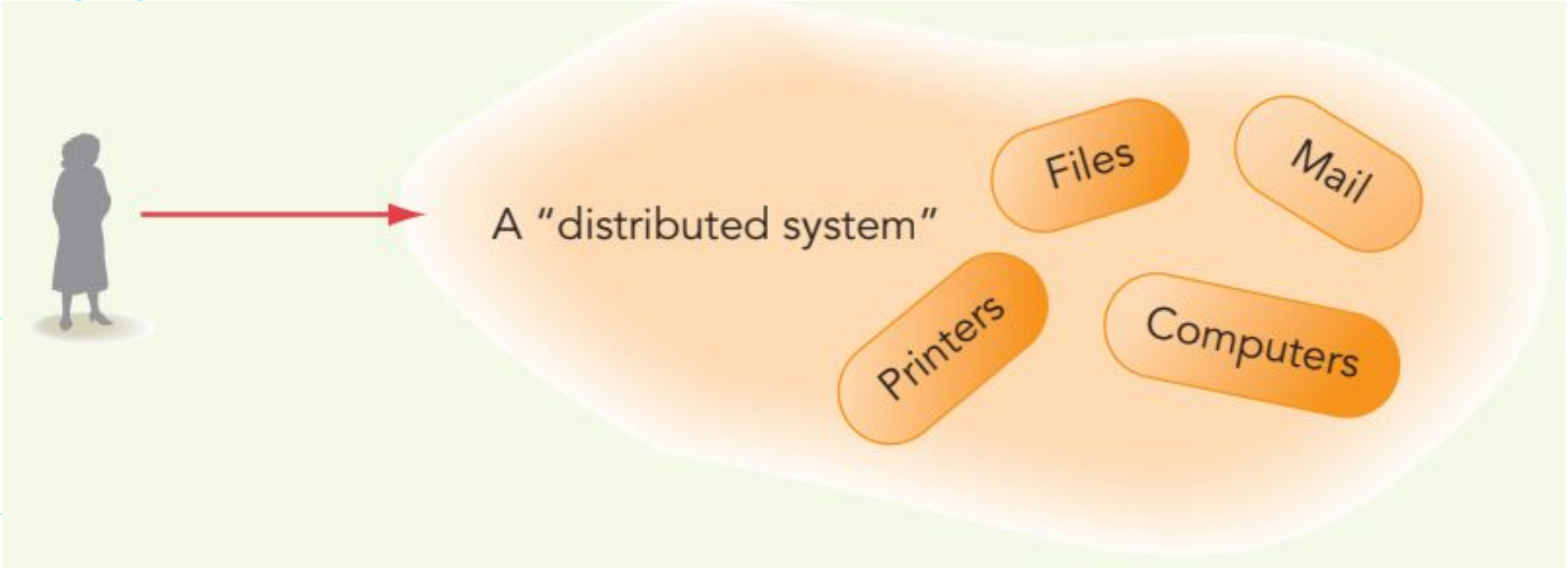
- A **distributed operating system** is system software over a collection of independent software, networked, communicating, and physically separate computational nodes handling jobs serviced by multiple CPUs.
- Each individual node holds a specific software subset of the global aggregate operating system.
- Each subset is a composite of two distinct service provisioners.
  - Microkernel directly controls the node's hardware.
  - A higher-level collection of *system management components* coordinating the node's individual and collaborative activities.
  - These components abstract microkernel functions and support user applications.

Distributed Operating System Architecture

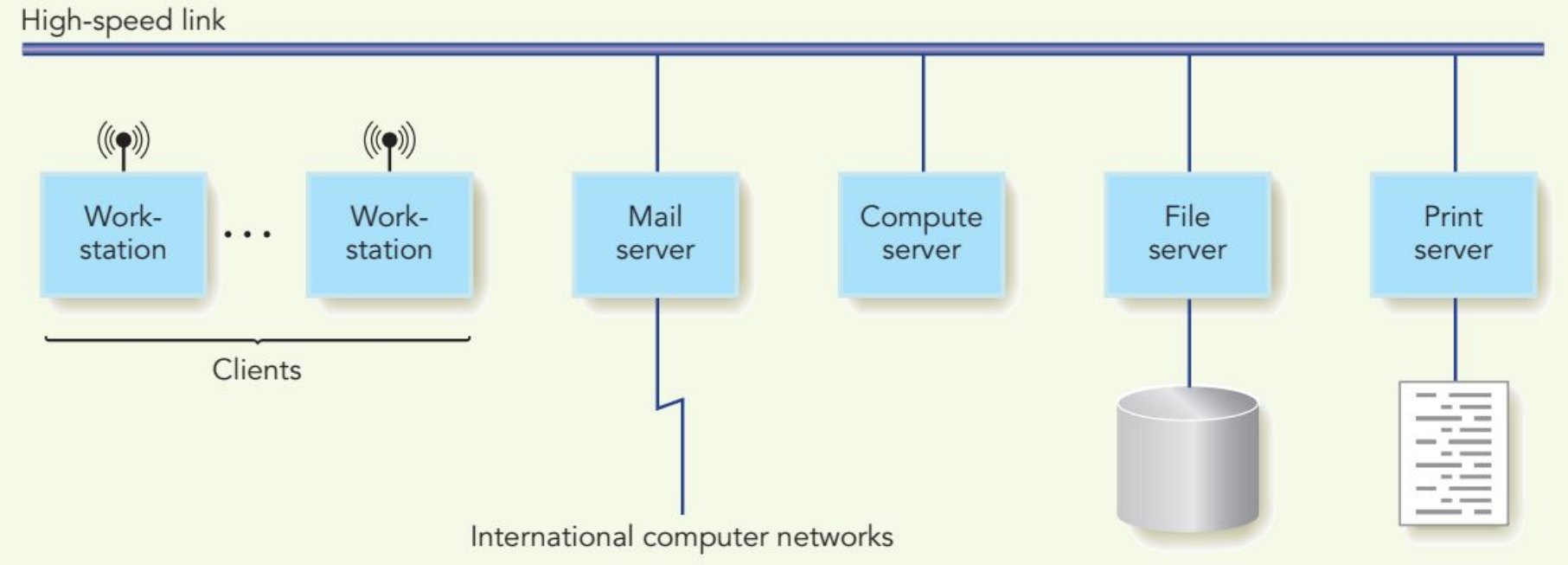




Virtual Environment of Distributed System

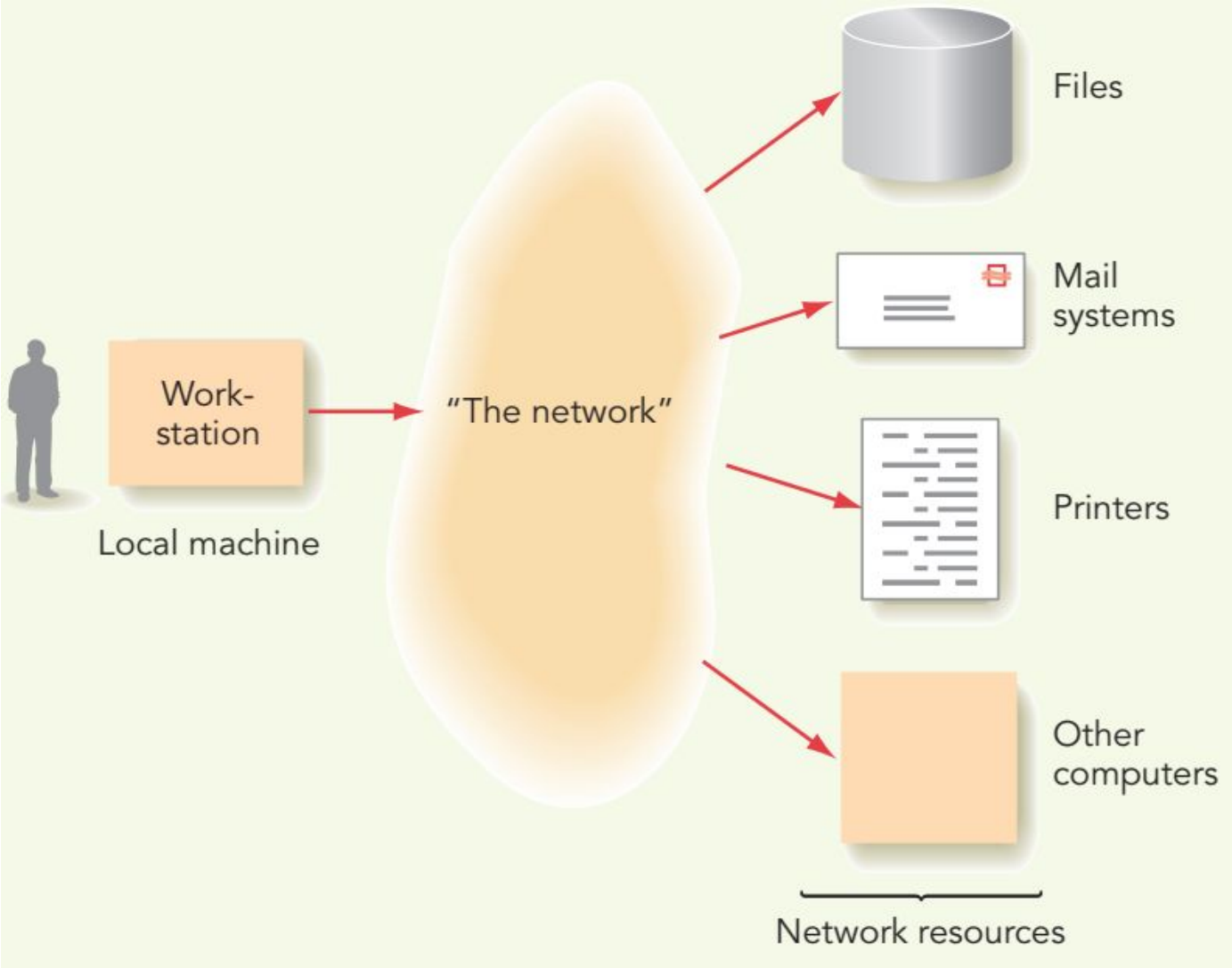


Network Operating System (4th Generation Operating System)



- A network operating system manages not only the resources of a single computer but also the capabilities of a telecommunications system called a *local area network* (*LAN*).
- A user can access local resources any one of the shared network resources just as though it were local.

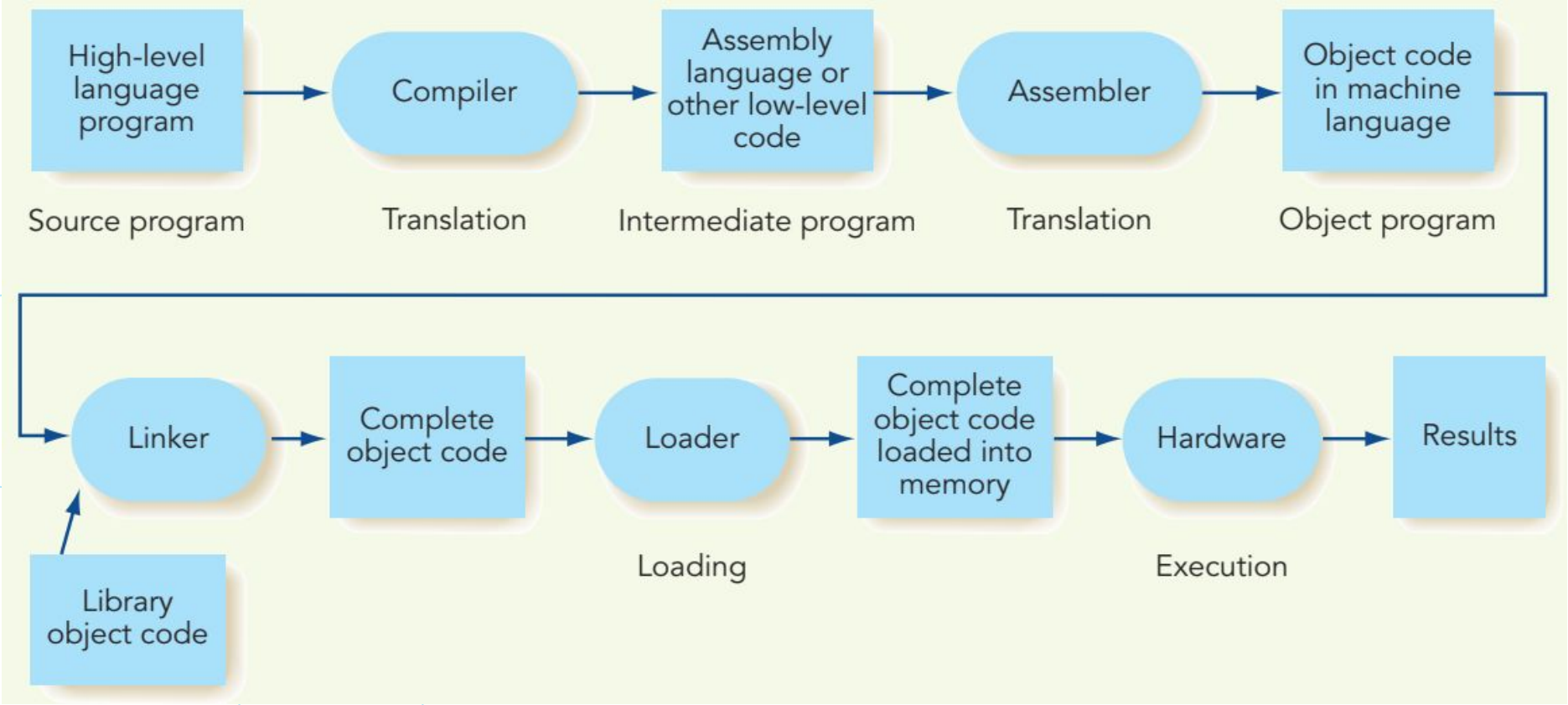
Virtual Environment



Evolution of Operation System

Generation	Approximate Dates	Major Advances
First	1945–1955	No operating system available Programmers operated the machine themselves
Second	1955–1965	Batch operating systems Improved system utilization Development of the first command language
Third	1965–1985	Multiprogrammed operating systems Time-sharing operating systems Increasing concern for protecting programs from damage by other programs Creation of privileged instructions and user instructions Interactive use of computers Increasing concern for security and access control First personal computer operating systems
Fourth	1985–present	Network operating systems Client-server computing Remote access to resources Graphical user interfaces Real-time operating systems Embedded systems
Fifth	??	Multimedia user interfaces Massively parallel operating systems Distributed computing environments

Transition of High-level Language Program



High Level Language Program Examples

1. Get value for the user's favorite number, *n*
2. Increase *n* by 1
3. Print a message and the new value of *n*

```
--Ada program for the
--favorite number algorithm

WITH TEXT_IO;

PROCEDURE FavoriteNumber IS
  PACKAGE INT_IO IS NEW TEXT_IO.INTEGER_IO(INTEGER);

        n : INTEGER;            --user's favorite number

BEGIN
  --Get the user's favorite number
  TEXT_IO.PUT("What is your favorite number? ");
  INT_IO.GET(n);

  --Compute the next number
  n := n + 1;

  --Write the output
  TEXT_IO.NEW_LINE;
  TEXT_IO.PUT("My favorite number is 1 more than that, ");
  INT_IO.PUT(n, 4);
  TEXT_IO.NEW_LINE;
  TEXT_IO.NEW_LINE;
END FavoriteNumber;
```

```
//C++ program for the
//favorite number algorithm

#include <iostream>
using namespace std;

void main()
{
     int n;    //user's favorite number

     //get the user's favorite number
     cout << "What is your favorite number? ";
     cin  >> n;

     //compute the next number
     n = n + 1;

     //write the output
     cout << endl;
     cout << "My favorite number is 1 more than that, "
          << n << endl;
     cout << endl << endl;
}
```

```
//C# program for the
//favorite number algorithm

using System;

namespace InvitationCSharp

{
    class FavoriteNumber
    {
        static void Main(string[] args)
        {
            int n;  //user's favorite number

            //get the user's favorite number
            Console.Write("What is your favorite number? ");
            n = Convert.ToInt32(Console.ReadLine());

            //compute the next number
            n = n + 1;

            //write the output
            Console.WriteLine();
            Console.Write("My favorite number is ");
            Console.WriteLine("1 more than that, " + n);
            Console.WriteLine();
            Console.WriteLine();
        }
    }
}
```

```
//Java program for the
//favorite number algorithm

import java.util.*;
public class FavoriteNumber
{
    public static void main(String[] args)
    {
        int n;          //user's favorite number
        Scanner inp = new Scanner(System.in);    //to read input

        //get the user's favorite number
        System.out.print("What is your favorite number? ");
        n = inp.nextInt();

        //compute the next number
        n = n + 1;

        //write the output
        System.out.println();
        System.out.println("My favorite number is 1 more "
          + "than that, " + n);
        System.out.println();
        System.out.println();
    }
}
```

```
#Python program for the
#favorite number algorithm

#get the user's favorite number
n = int(input("What is your favorite number? "))

#compute the next number
n = n + 1

#write the output
print()
print("My favorite number is 1 more than that,", n)

#finish up
input("\n\nPress the Enter key to exit");
```

Features

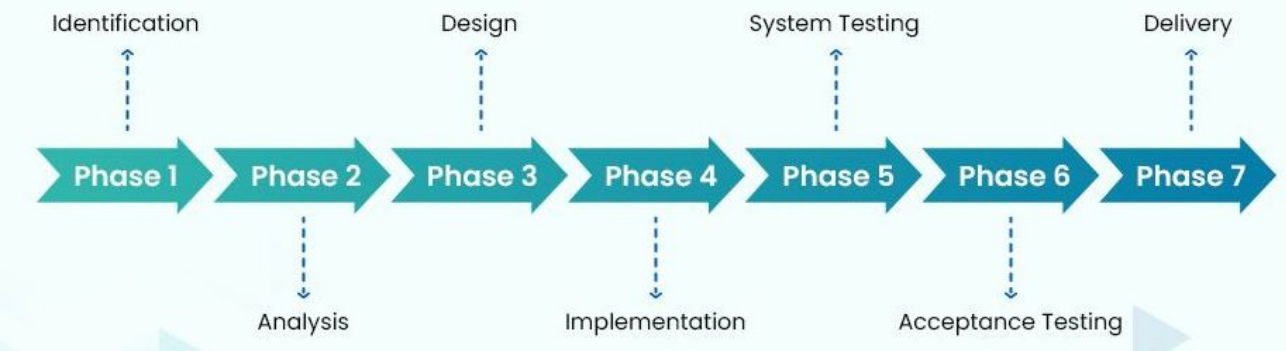
- *The programmer need not manage the details of the movement of data items within memory or pay any attention to exactly where those items are stored*.
- *The programmer can take a macroscopic view of tasks, thinking at a higher level of problem solving*.
- *Programs written in a high-level language will be portable rather than machine specific*.
- *Programming statements in a high-level language will be closer to natural language and will use standard mathematical notation*.

Scale Up

Category	Typical Number of People	Typical Duration	Product Size in Lines of Code	Examples	Building Analogy
Trivial	1	1–2 weeks	< 500	Student homework assignments	Small home improvement
Small	1–3	A few weeks or months	500–2,000	Student team projects, advanced course assignments	Adding on a room
Medium	2–5	A few months to 1 year	2,000–10,000	Research projects, simple production software such as assemblers, editors, recreational and educational software	Single-family house
Large	5–25	1–3 years	10,000–100,000	Most current applications—word processors, spreadsheets, operating systems for small computers, compilers, iPhone apps	Small shopping mall
Very Large	25–100	3–5 years	100,000–1 M	Airline reservations systems, inventory control systems for multinational companies	Large office building
Extremely Large	> 100	> 5 years	> 1 M	Large-scale real-time operating systems, advanced military work, international telecommunications networks	Massive skyscraper

Software Development Life Cycle

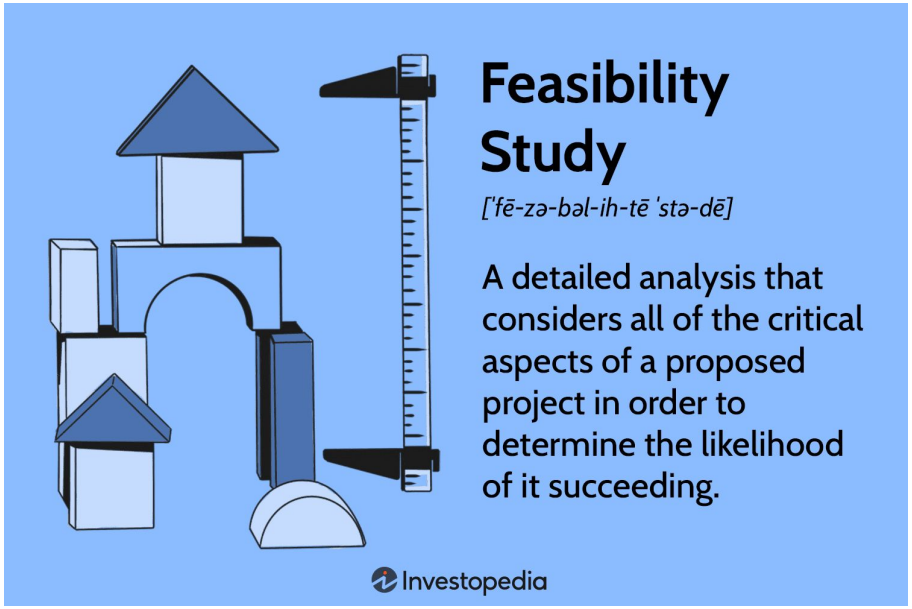
1. Before Implementation
   a. Feasibility study
   b. Problem specification
   c. Program design
   d. Algorithm selection or development, and analysis
2. Implementation
   a. Coding
   b. Debugging
3. After Implementation
   a. Testing, verification, and benchmarking
   b. Documentation
   c. Maintenance



Feasible Study

- The **feasibility study** evaluates a proposed project and compares the costs and benefits of various solutions.
- Buying a new computer system and writing or buying software
- Writing new software for an existing computer system
- Purchasing the needed resources from a “cloud computing” provider
- Outsourcing the work to a contractor
- Revising the current manual process for solving this problem
- Cutting back the scope of the project to better align it with existing resources
- Cancelling the project entirely and doing without the information that would be generated

→ feasibility document



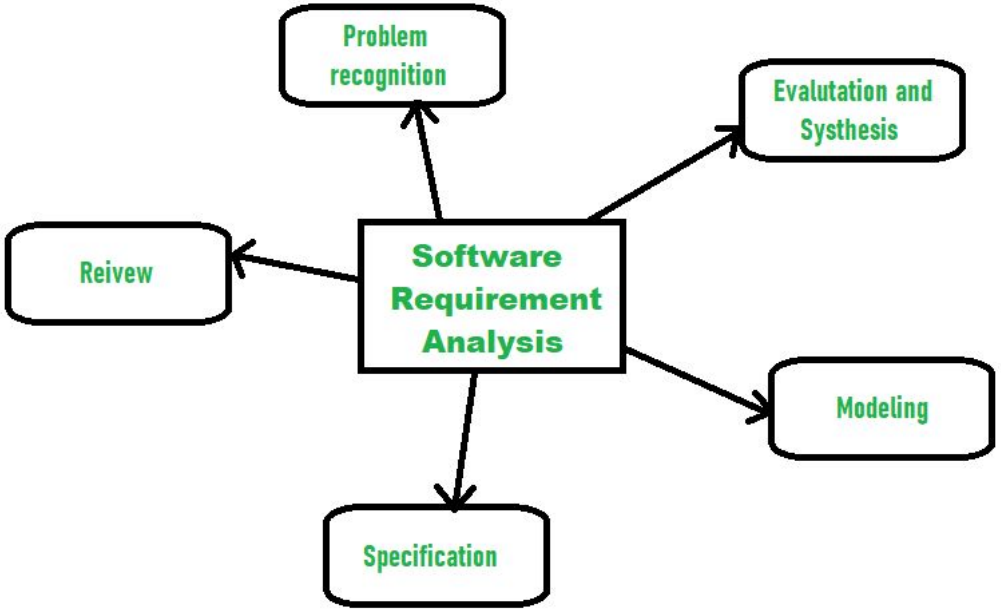
Feasible Study

- **Problem specification** involves developing a clear, concise, and unambiguous statement of the exact problem the software is to solve.

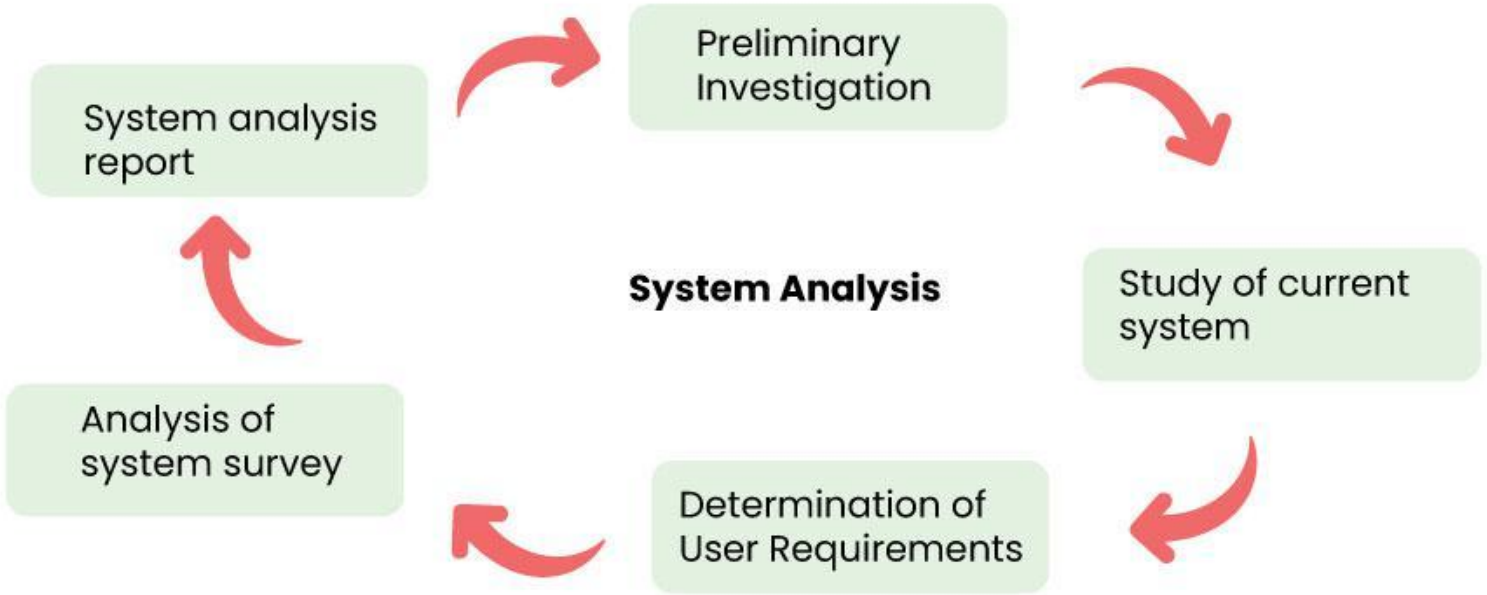
→ *problem specification*

- **Requirement Analysis:** the process of determining user expectations for a new or modified product

→ *specific requirement*



System Analysis



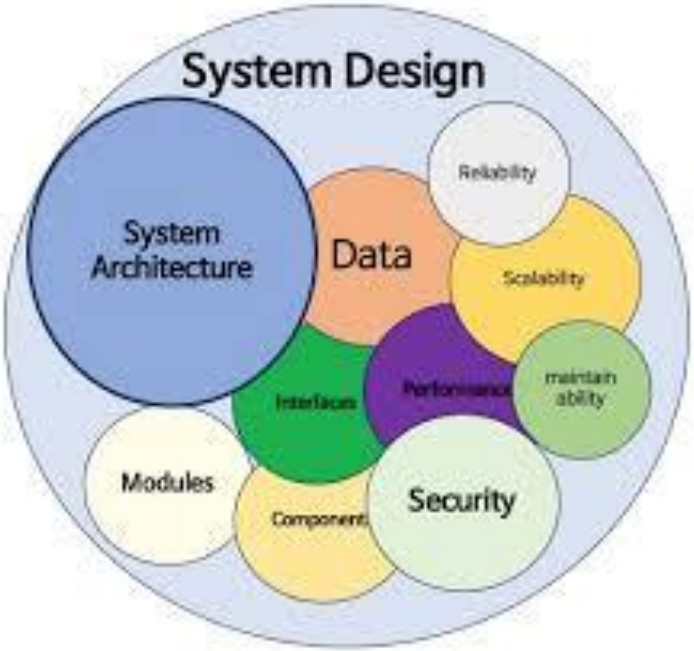
System Analysis (System Analysis | System Design)

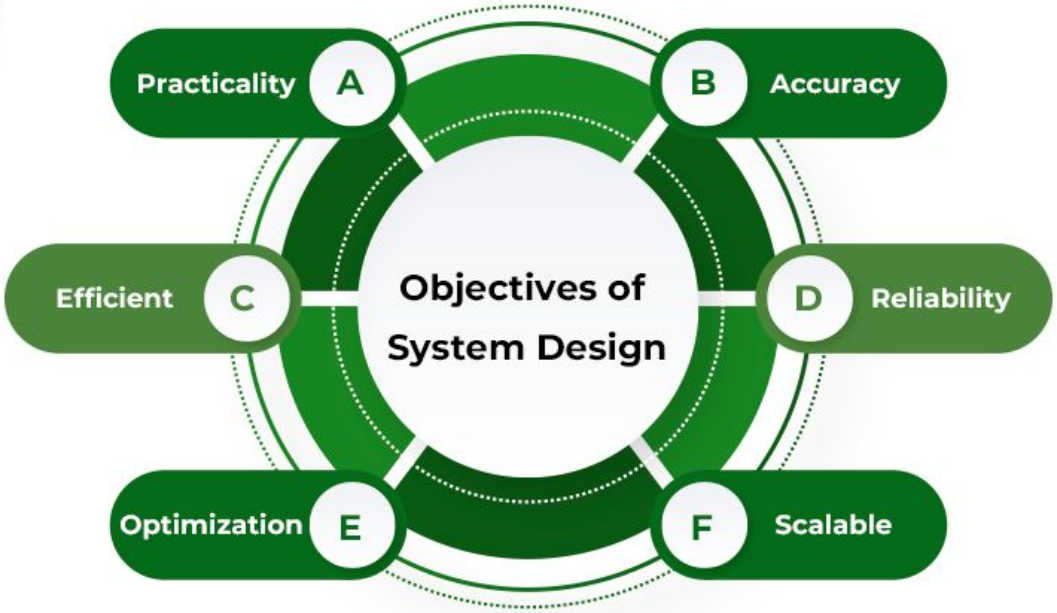
Program Design

- *Program design*— plan *how the* project is to be done.
  - **Divide-and-conquer** strategy (**top-down decomposition**): Tasks are broken down into subtasks, into sub-subtasks, and so on, until each piece is small enough to code comfortably.
  - **Object-oriented programming** approach*:* the appropriate objects are identified, together with their data and the subtasks they must perform. This allows classes to be designed with variables to store the data, and functions (also called *methods*) to carry out these subtasks. Objects from these classes cooperate to accomplish the total job.

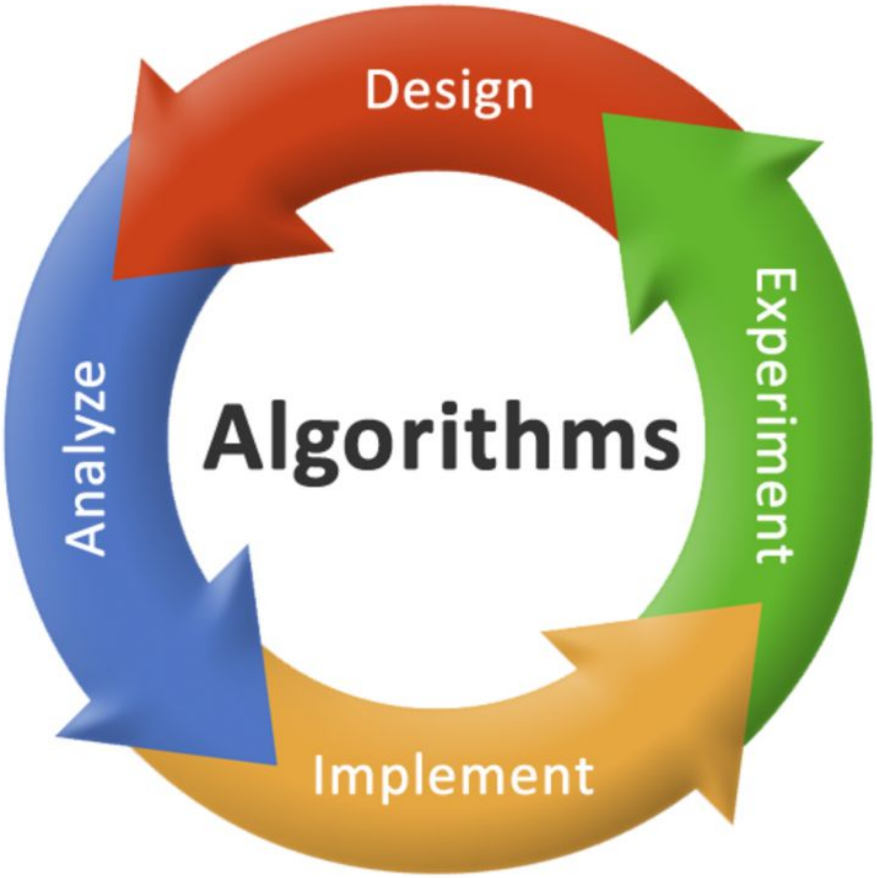
→ *program design document*

System Design



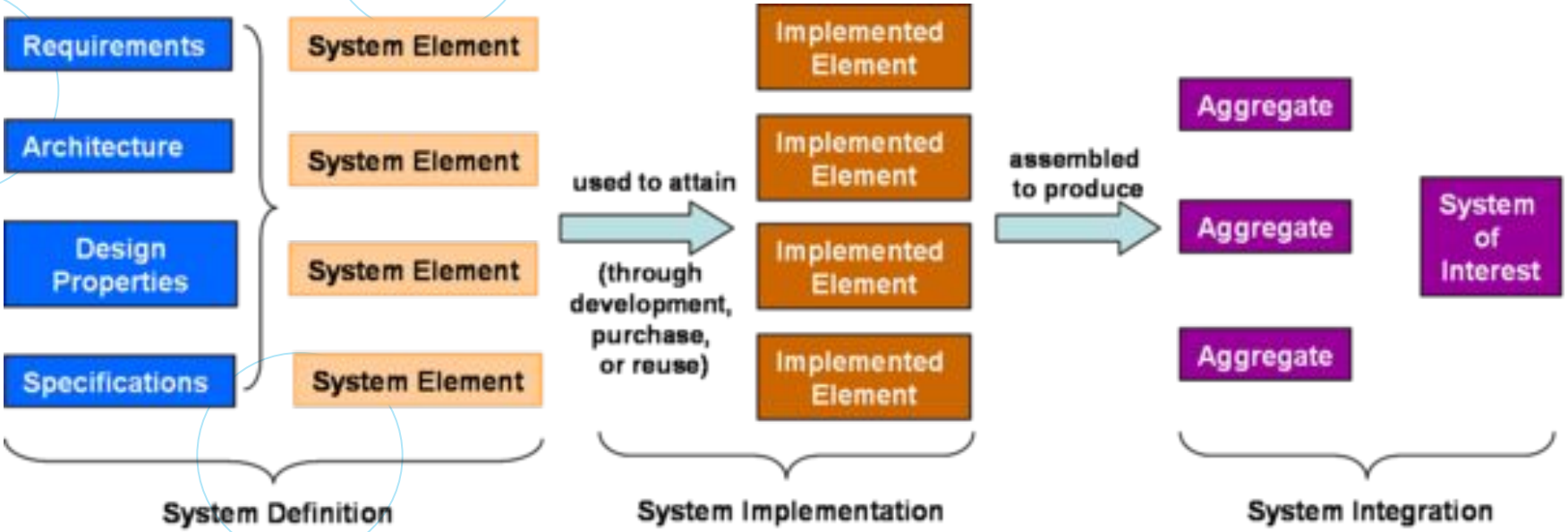


- *Algorithm selection or development, and analysis*



Problem Implementation

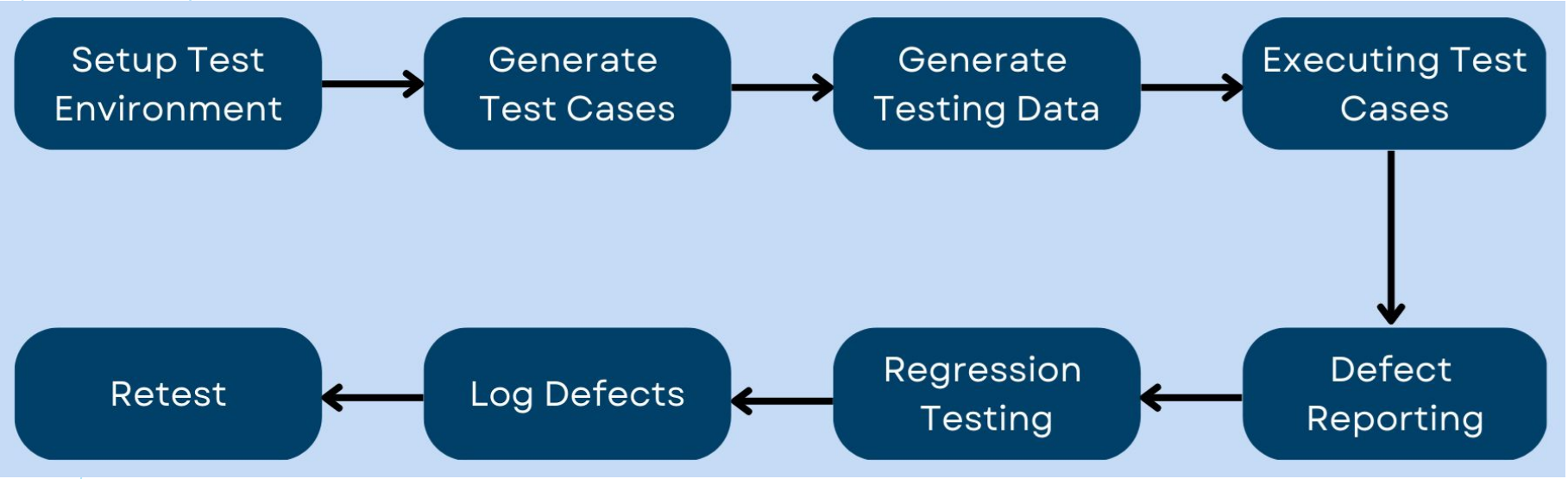
- **Coding**
- **Debugging**
  - Syntax Error
  - Runtime Error
  - Logic Error

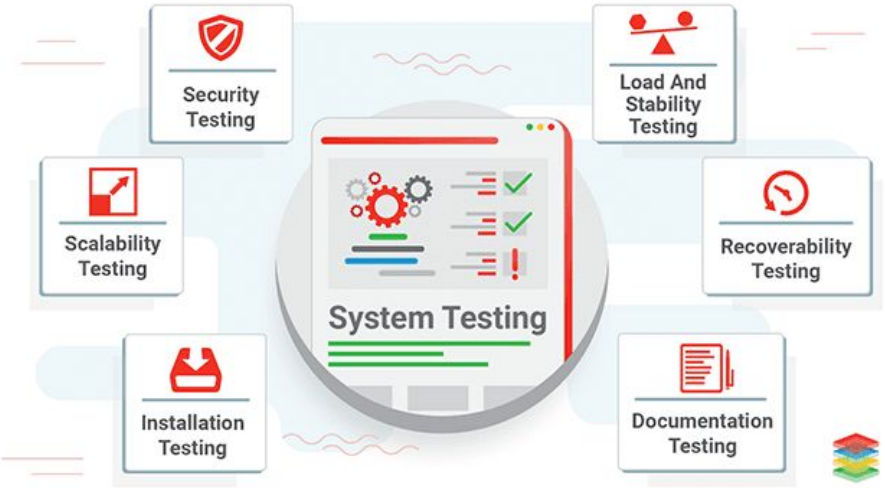


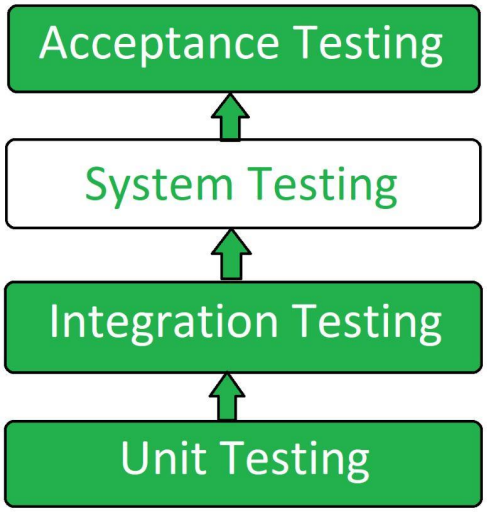
Testing, Verification, and Benchmarking

- **Empirical testing**: design a special set of test cases, **test suite**
- **Unit testing** takes place on each module (subtask code) as it is completed
- **Integration testing:** the modules communicate the necessary data between and among themselves and that all modules work together smoothly
- **Regression testing:** done on changed modules to be sure that this change hasn't introduced a new error into code that was previously correct
- **Program verification:** prove that if the input data to a program satisfies certain conditions, then the output data satisfies certain other conditions. This process is a formal proof, much like the proof of a mathematical theorem that condition B follows from condition A.
- *Benchmarking:* running the program on many data sets to be sure its performance falls within those required limits.

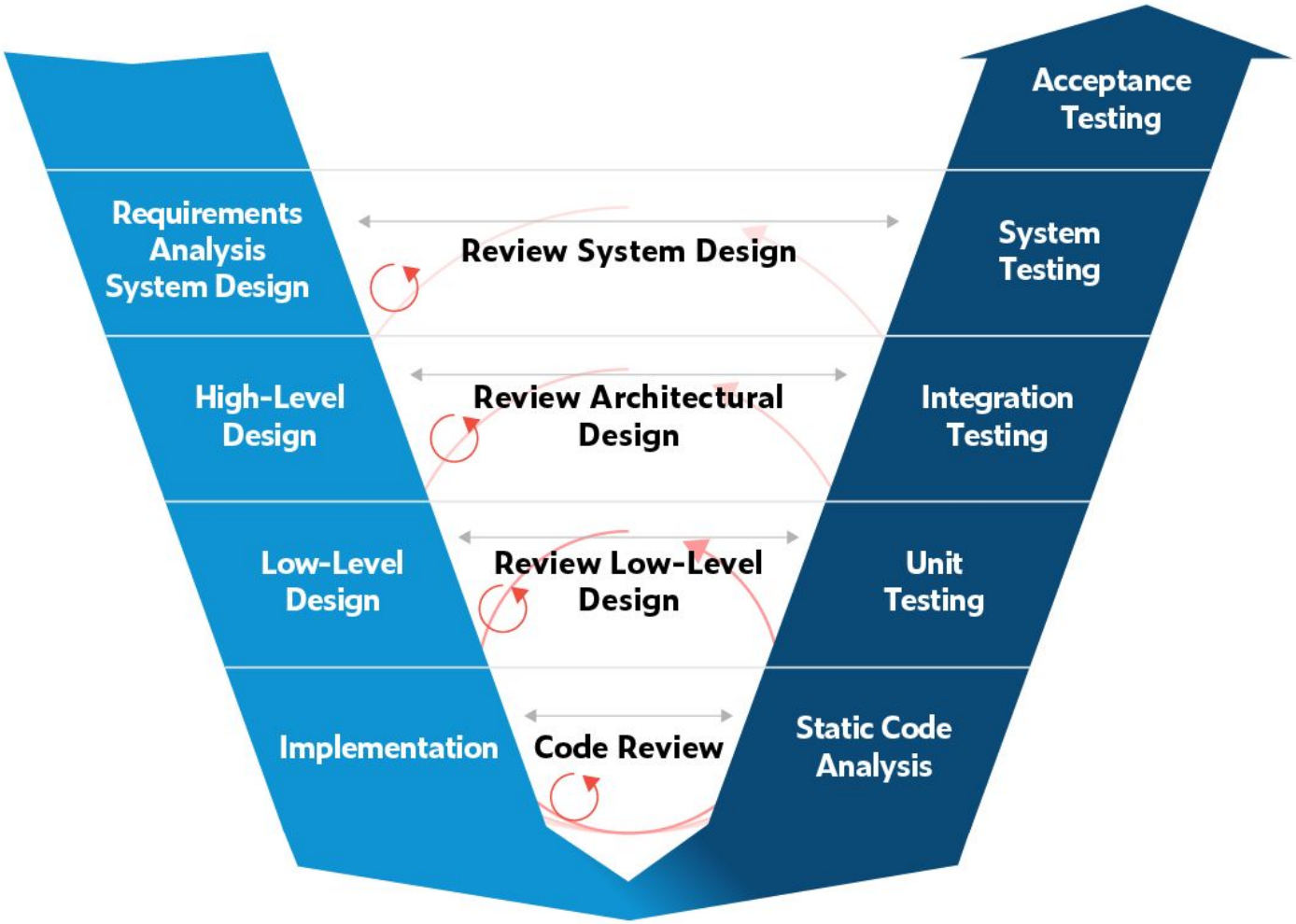
System Testing



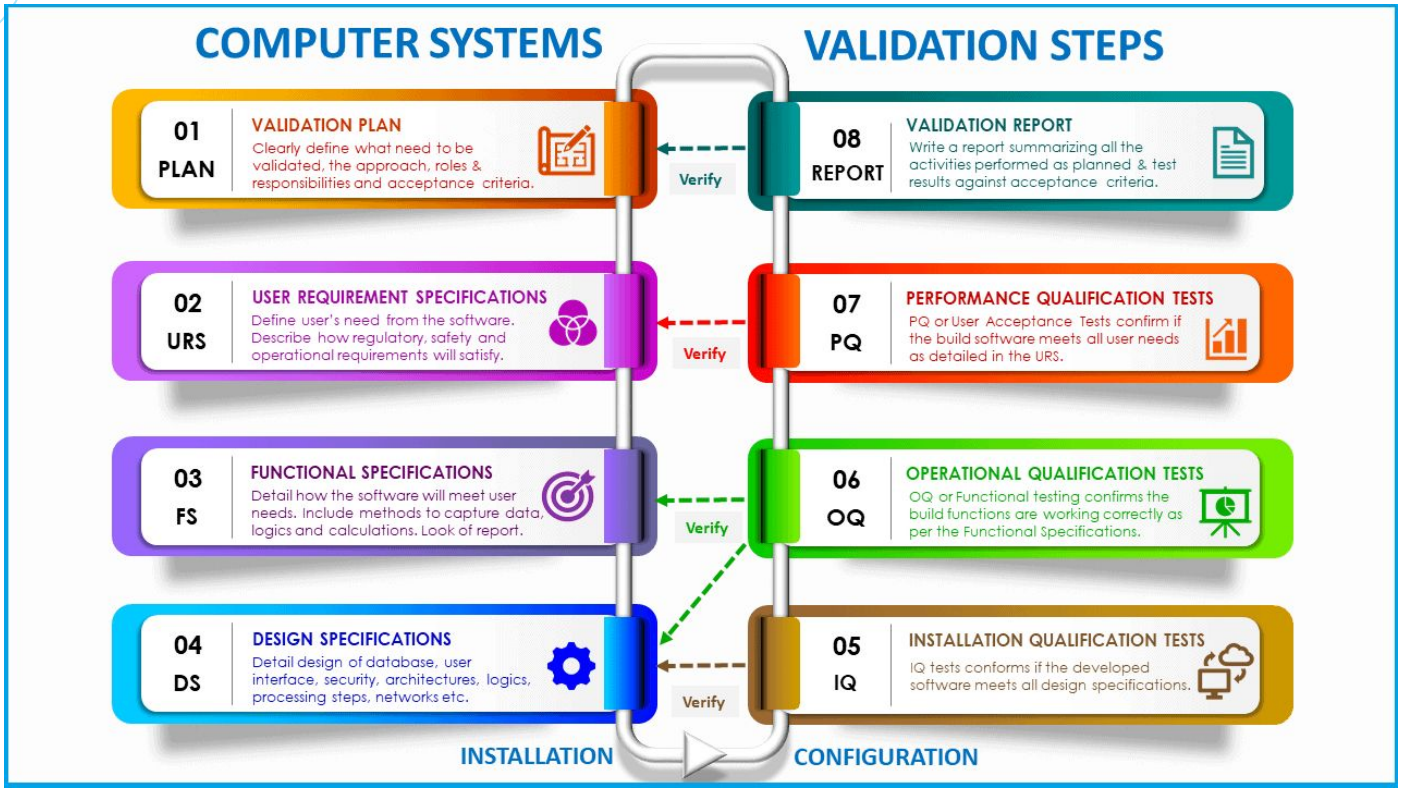


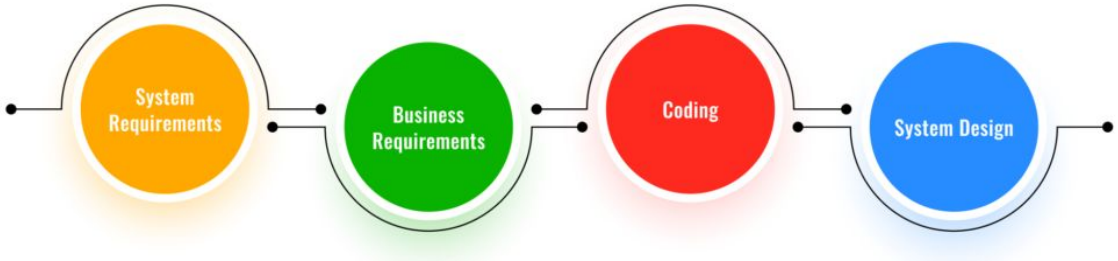


V-Model


Requirements Analysis System Design
High-Level Design
Low-Level Design
Implementation
Review System Design
Review Architectural Design
Review Low-Level Design
Code Review
Acceptance Testing
System Testing
Integration Testing
Unit Testing
Static Code Analysis

Software Verification and Validation



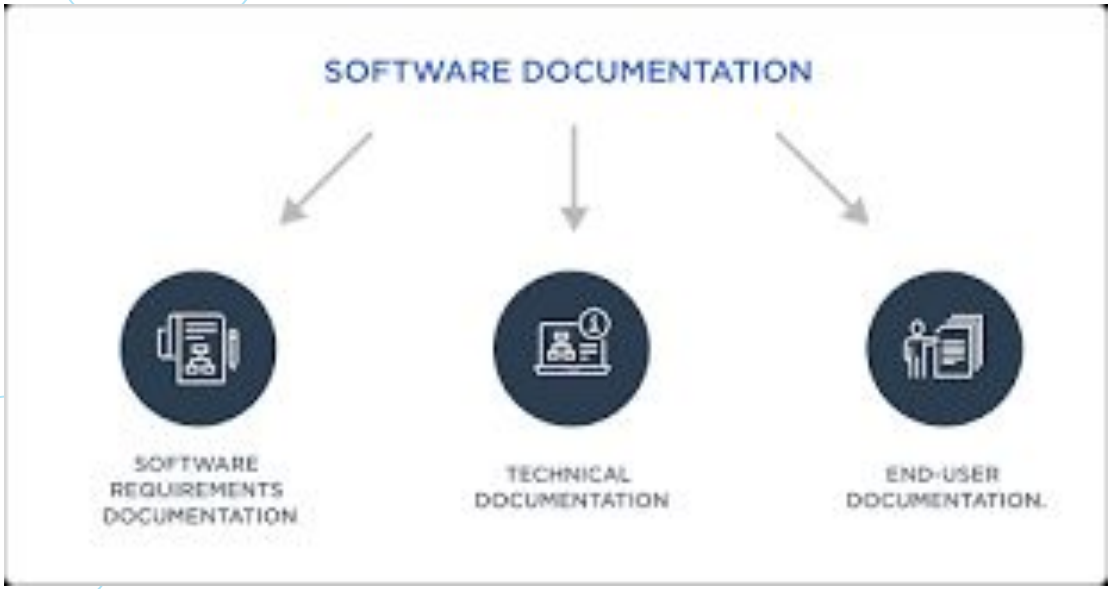


Documentation: Make the program understandable

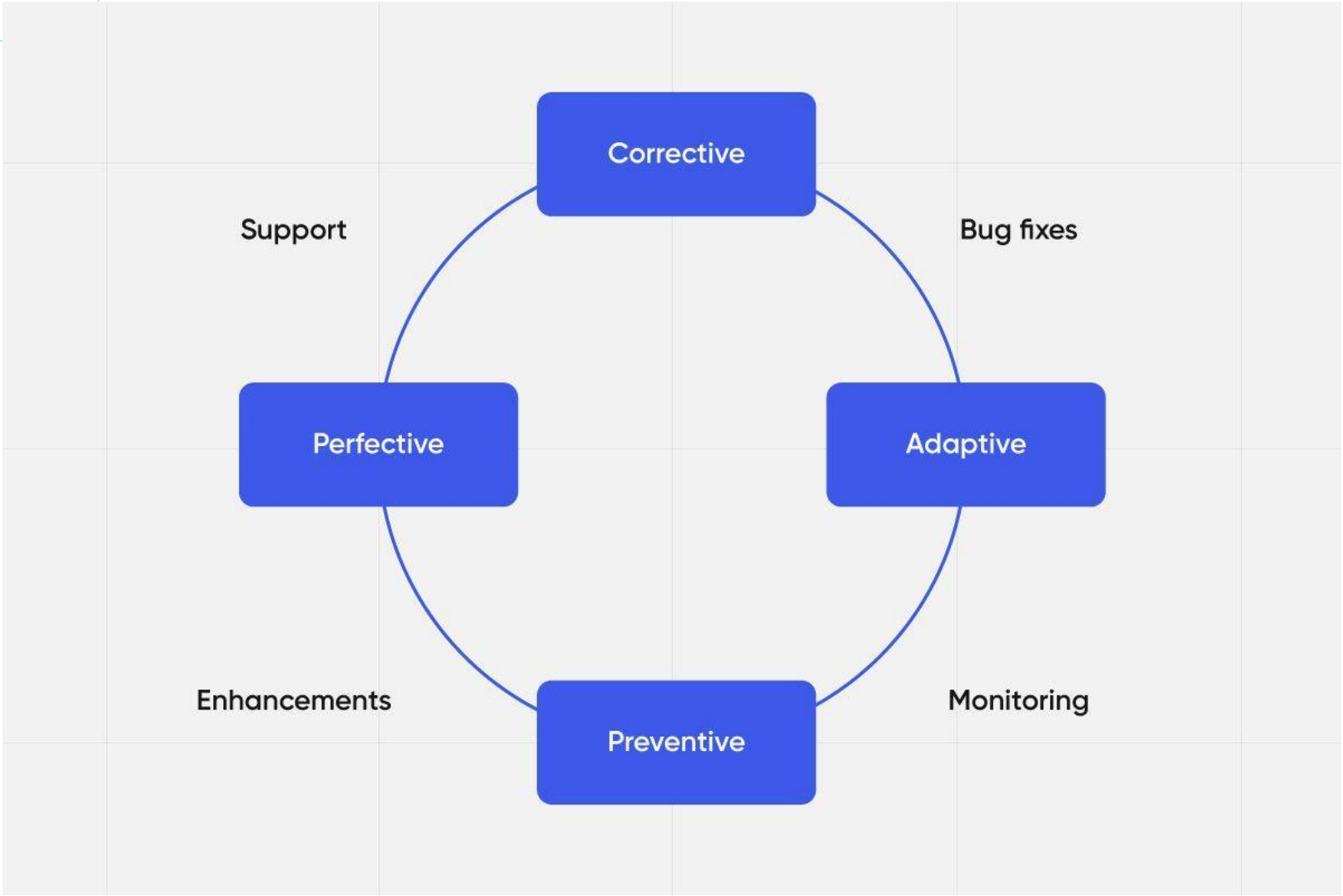
- **Internal documentation**: part of the program code
  - meaningful names for identifiers (constant, variable, procedure, function, …)
  - comments to explain the code,
  - Separation of the program into short modules
- **External documentation:** clarify the program's design and implementation
- **Technical documentation:** enables the code understandable
  - problem specification documents
  - design documents
  - structure charts or class diagrams
  - descriptions of algorithms
  - program listings

Documentation: Make the program understandable

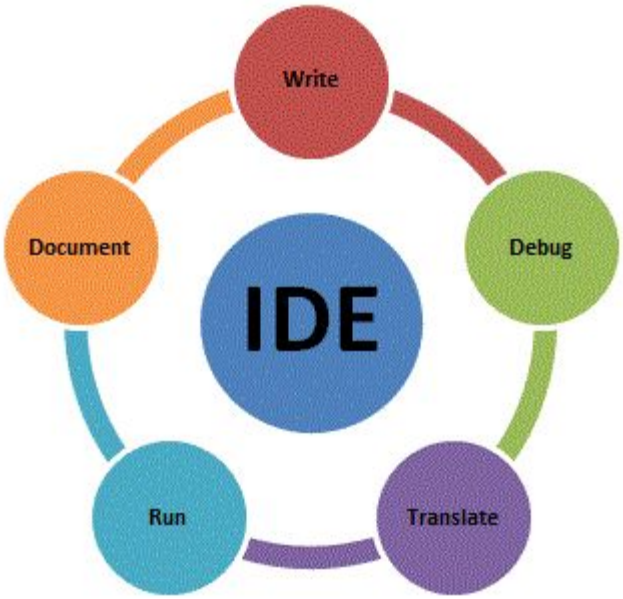
- **User documentation:** helps users run the program
  - online tutorials
  - answers to frequently asked questions (FAQs)
  - help systems
  - written users' manuals



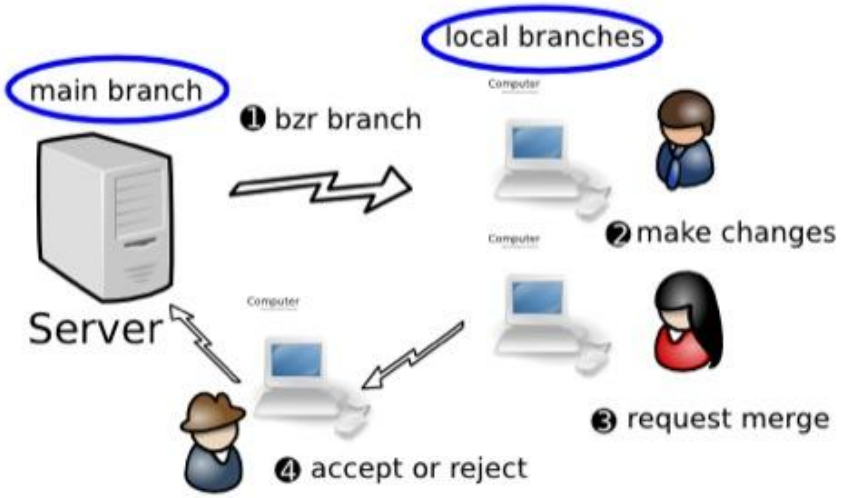
Maintenance



Integrated Development Environment



Version Control System



BENEFITS OF VERSION CONTROL

01 Streamline merging and branching

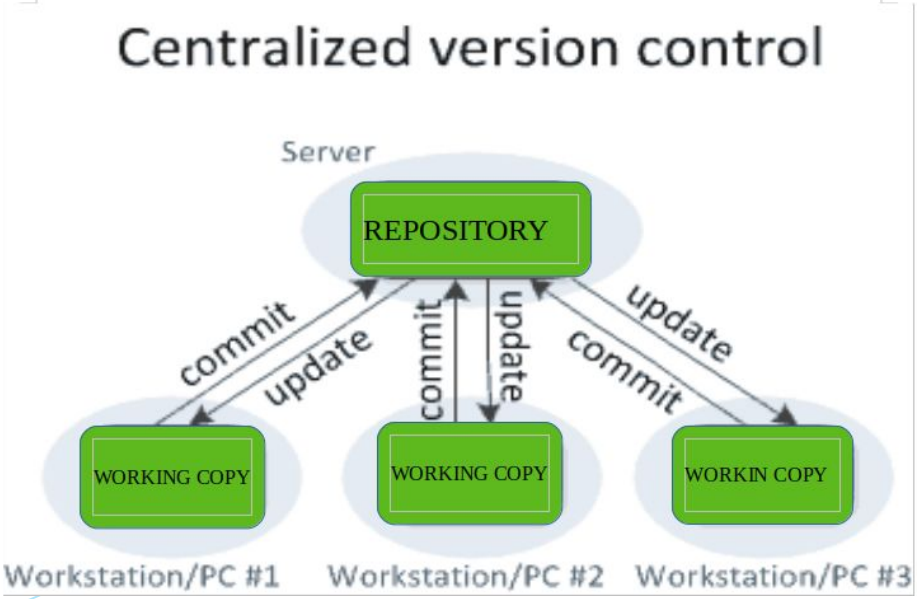
02 Examine and experiment with code

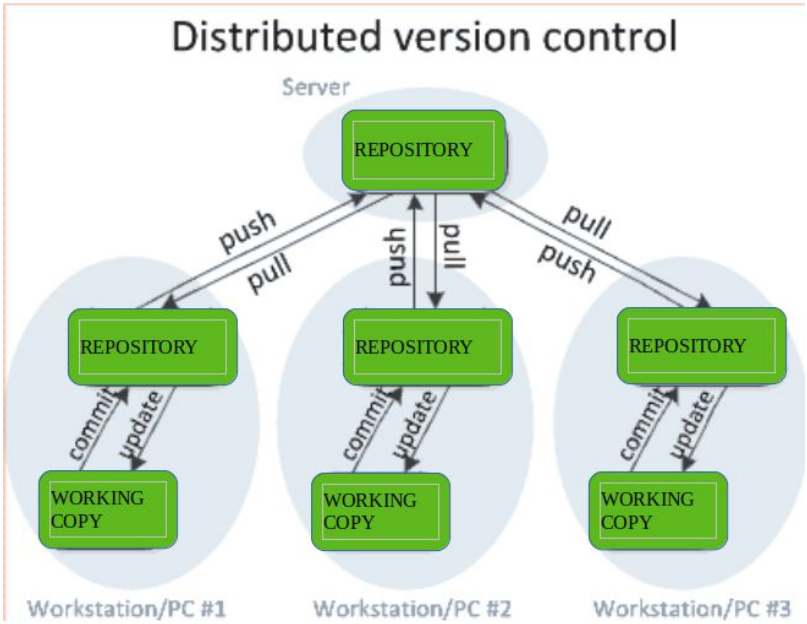
03 Discover the ability to operate offline

04 Create regular, automated backups

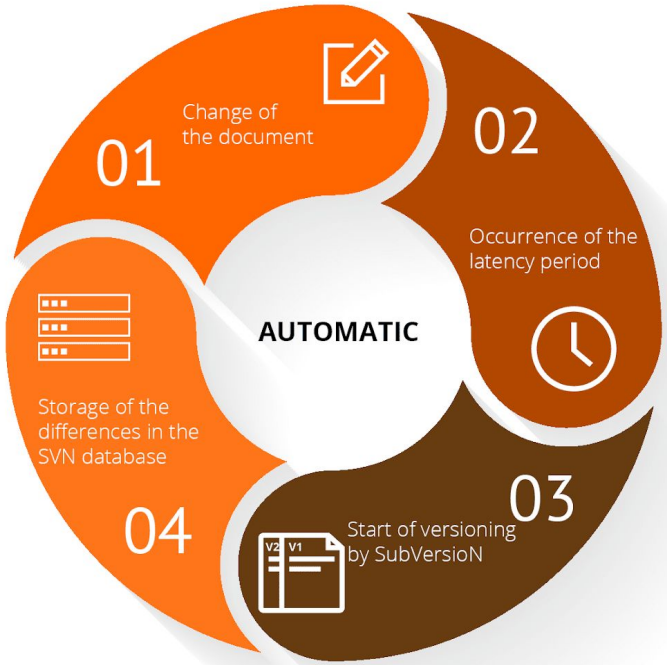
05 Communicate through open channels

Version Control System

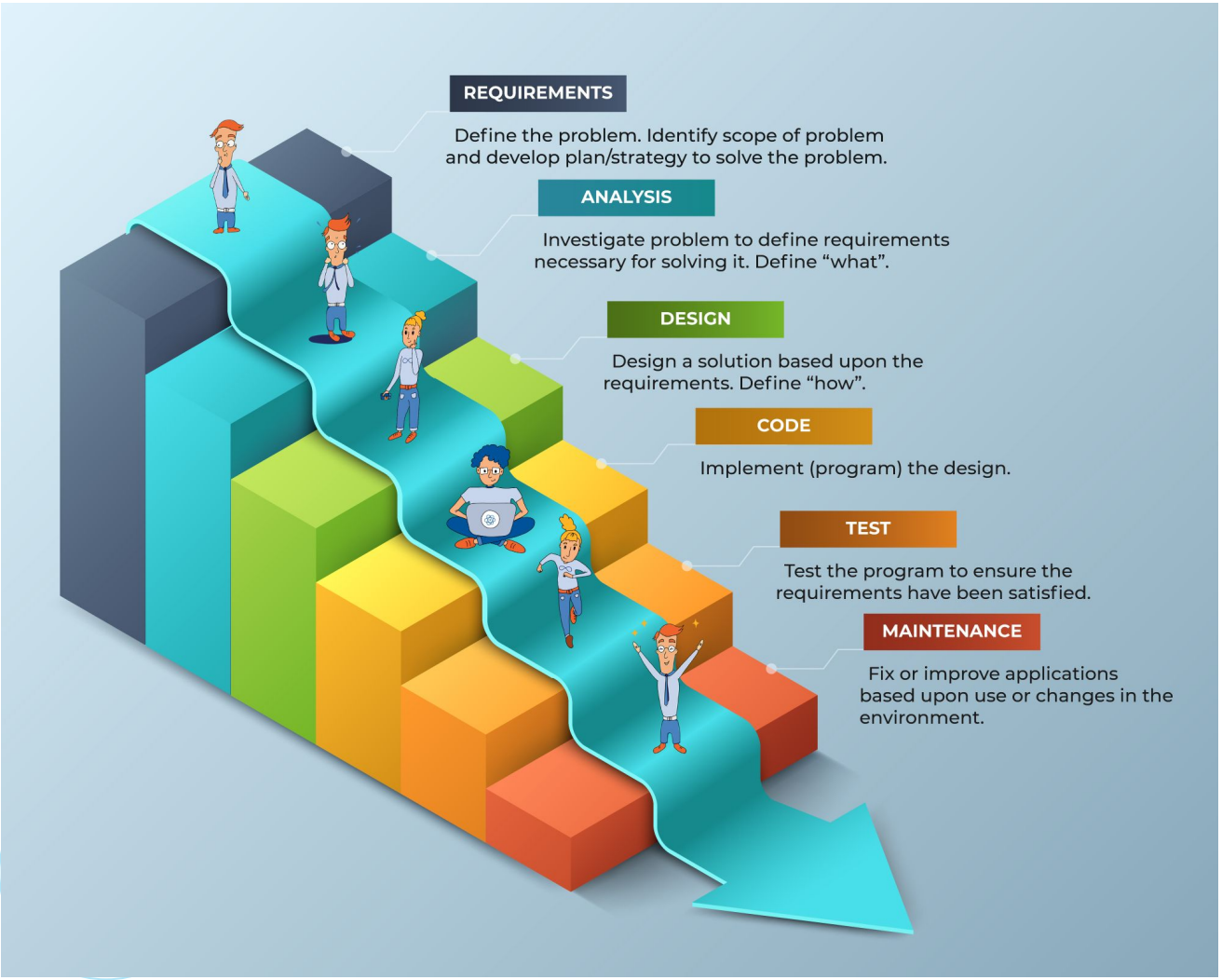




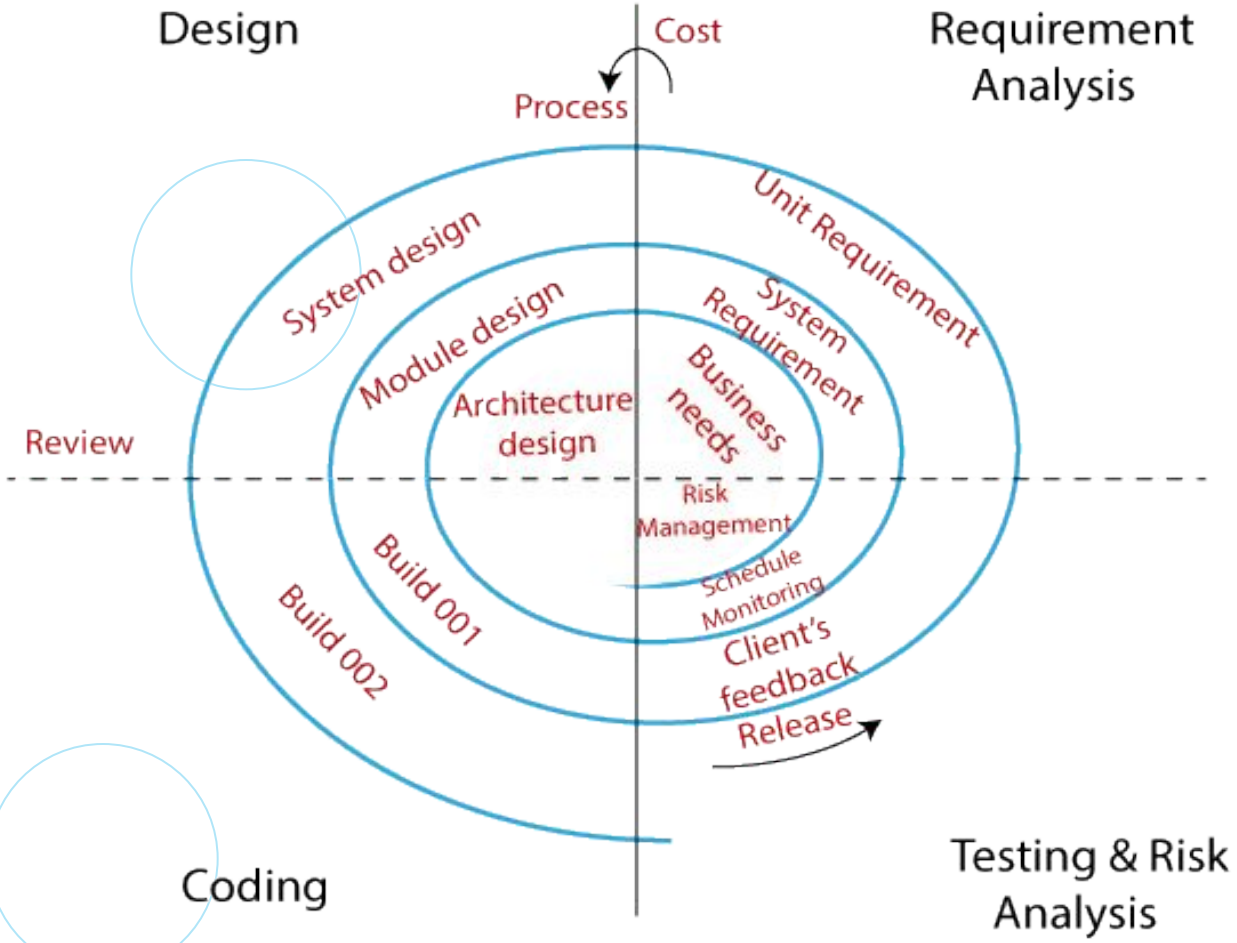
Version Control System



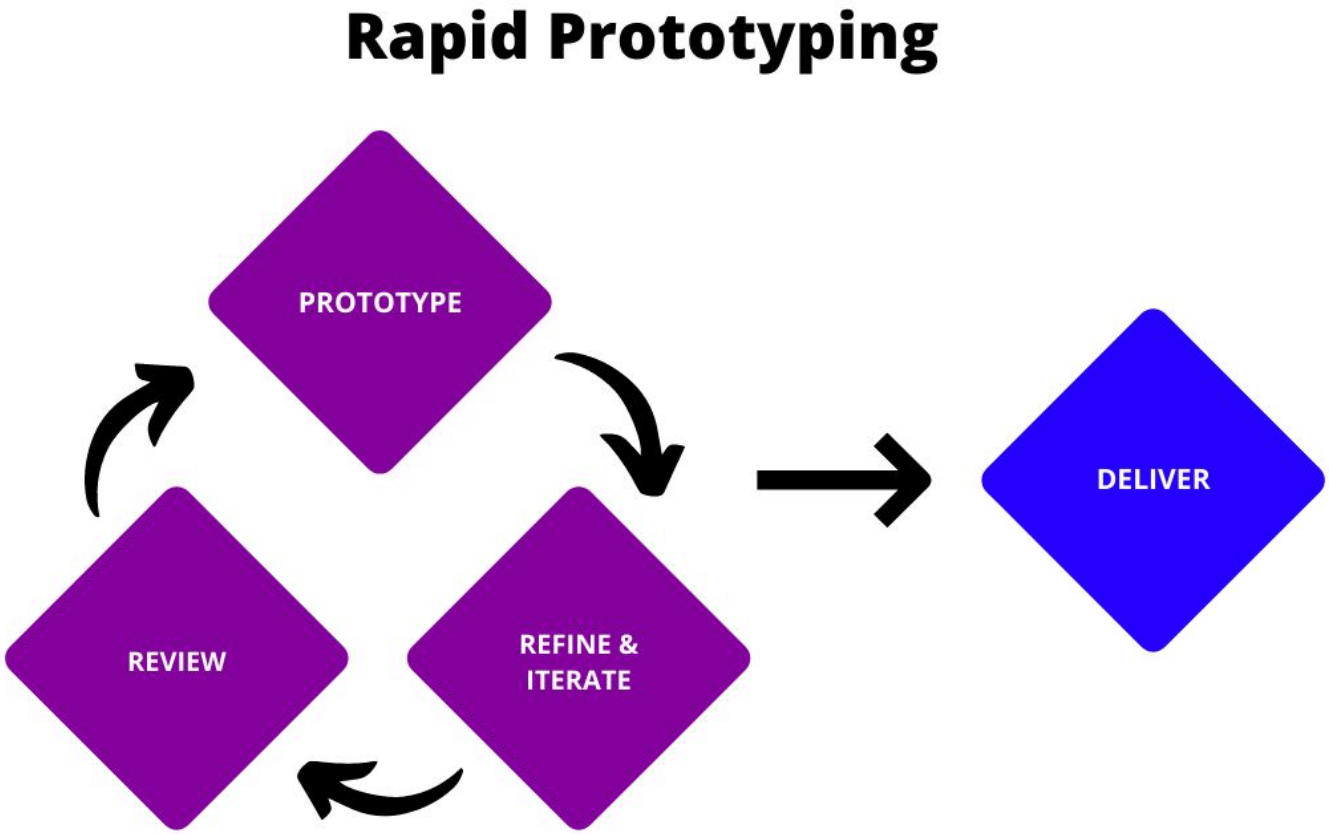
Waterfall Model



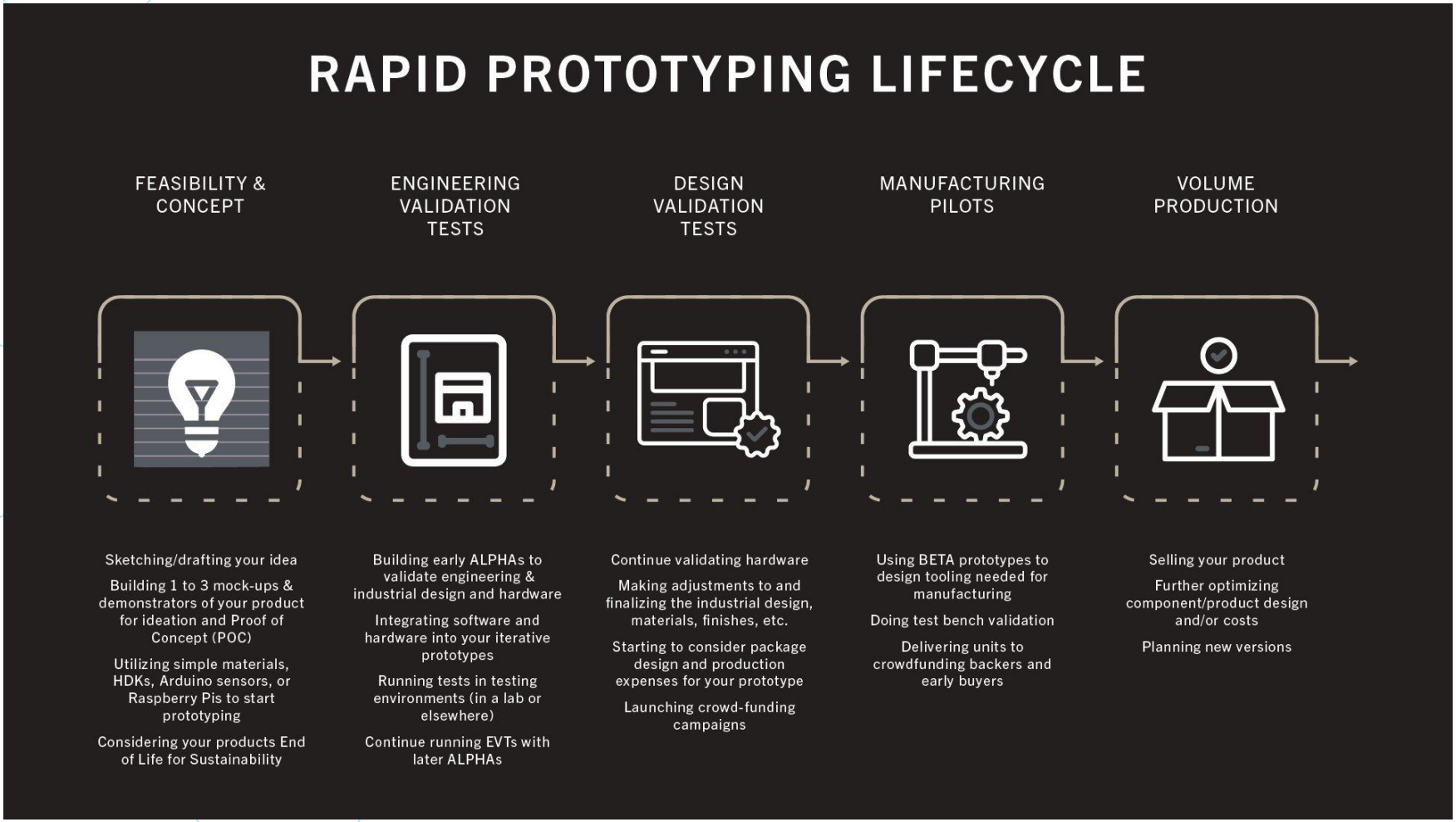
Spiral Model



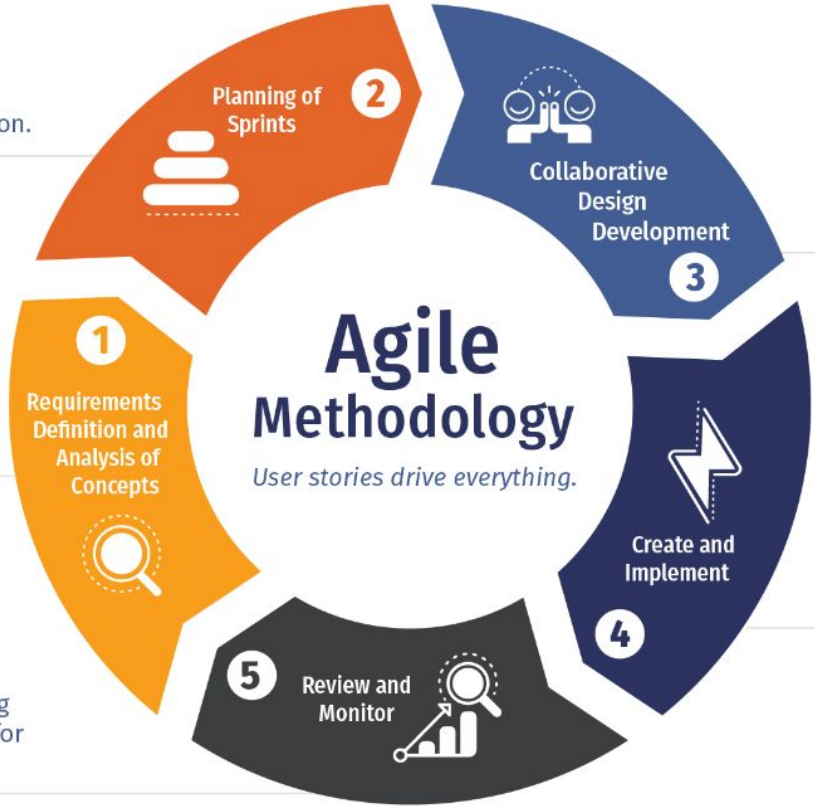
Rapid Prototyping Model



Rapid Prototyping Lifecycle



Agile Methodology



Analysis of concepts and requirements definitions; Determine current state and your expectations.

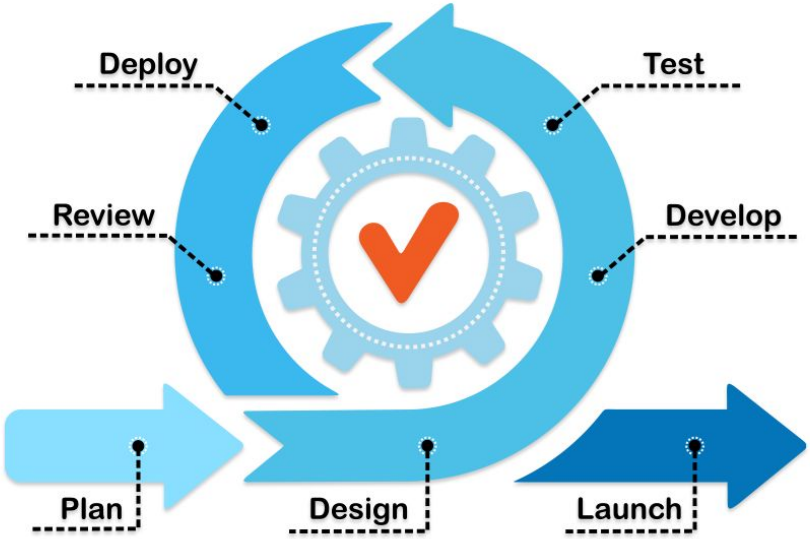
Arrange teams and tools needed to optimize production.

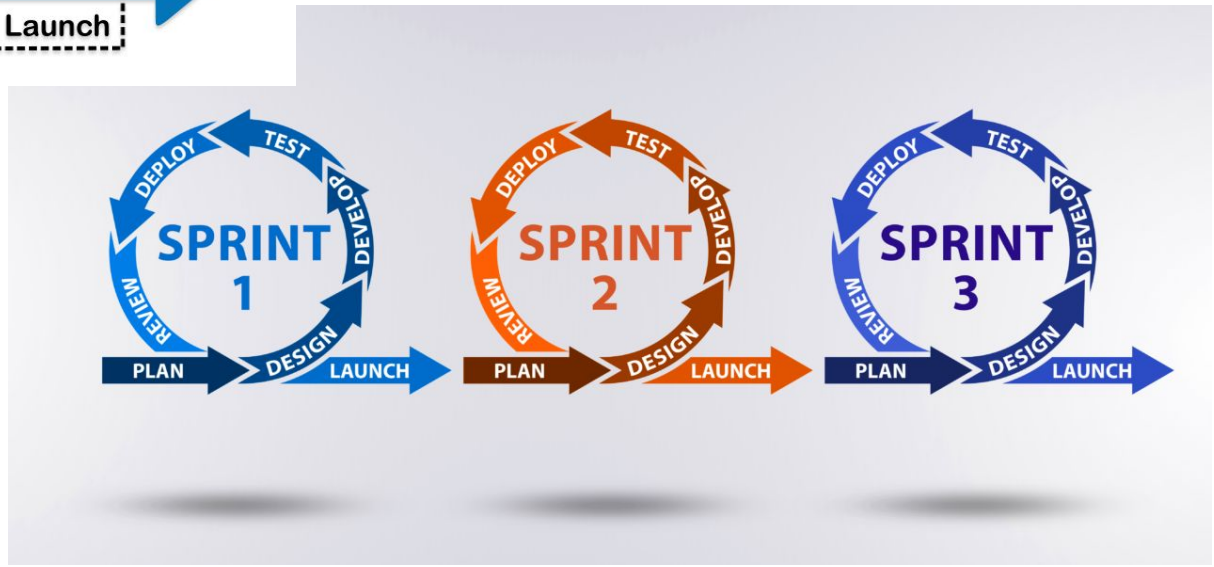
From the beginning of the process, the end users' involvement and feedback is critical.

Frequent development delivery through sprints. Feedback on testing & appropriate changes are imperative.

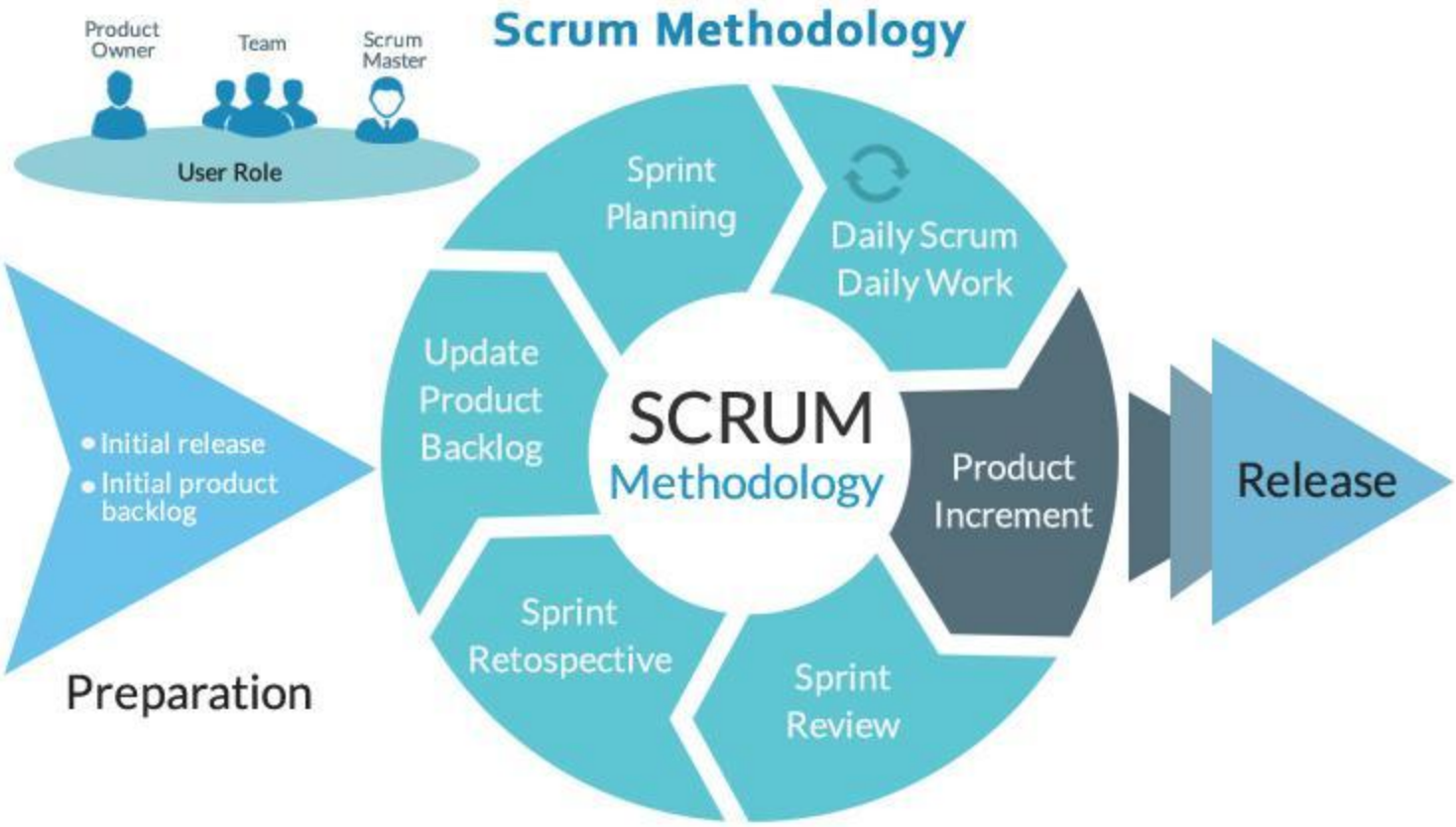
Ensure that you are reviewing and monitoring key metrics for success.

Agile Methodology

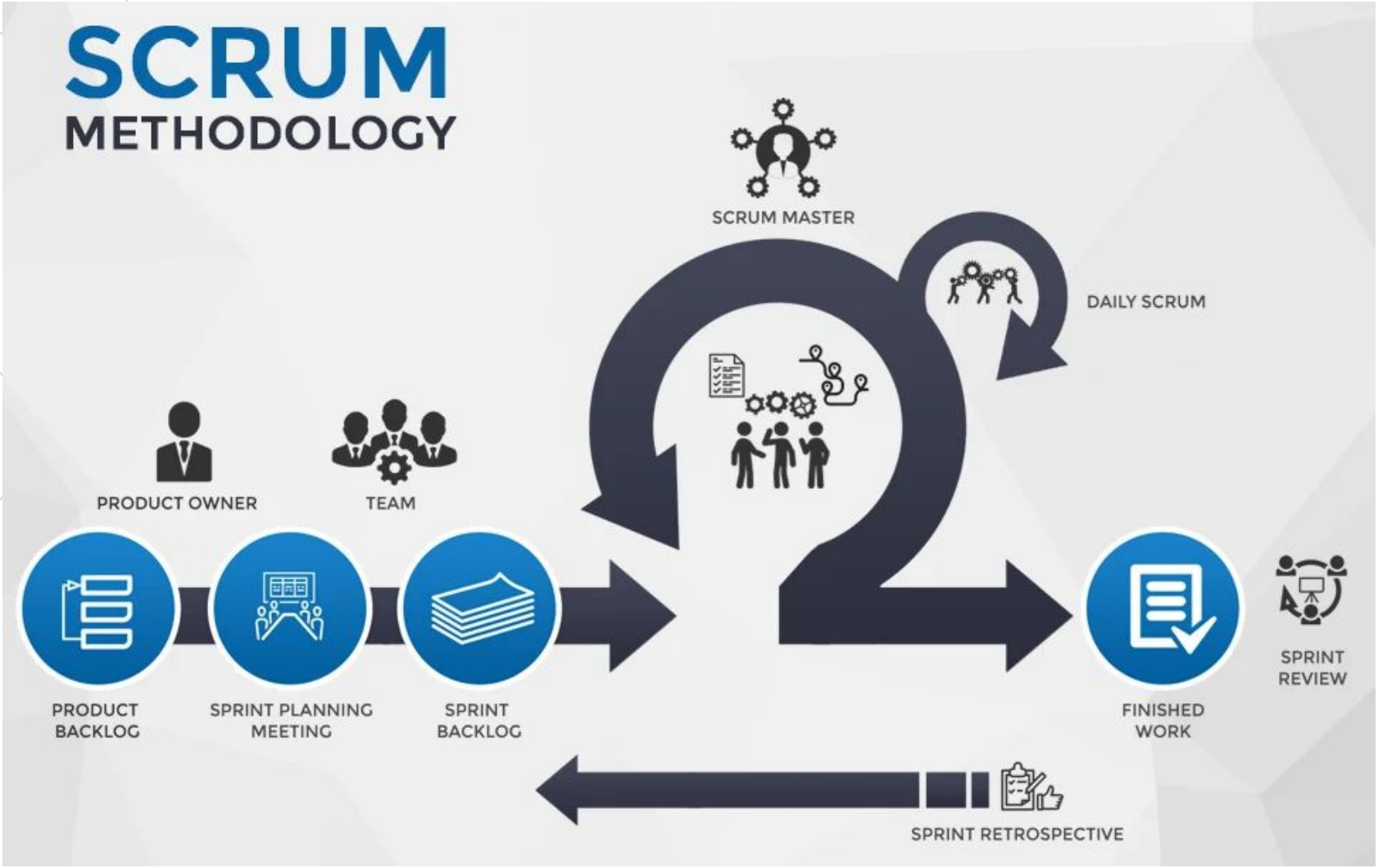




Scrum Methodology



Scrum Methodology



HUST

hust.edu.vn fb.com/dhbkhn

FaMI 1956