Design Methodologies

- From Complex problems to Simple problems.
- Top- down Design:
  - Directly into Modular Structure
    - Simple Tasks are used as abstract tools by superior modules to solve more complex problems in the system.
- Bottom-up Design:
  - Identifying individual tasks within the system.
  - Solutions to these tasks are used as abstract tools in the solution to more complex problem.
    - Better than Supervision: Example Parallel Processing Applications.
    - Off –the-shelf Components: Building complex S/W Sys.

Tools of the Trade

- Dataflow Diagram
- Entity-Relationship Diagram
  - Example: Professor, Classes and Students
  - Object-Oriented Design Environments
- Data dictionary
  - Avoiding misunderstanding
  - Reducing Redundancy and contradiction
- CRC (Class-Responsibility-Collaboration)
  - Traditional index card on which description of object is being written.

Design Patterns

- Inspiration from Architecture
- In 1977, Pattern Language by Christopher Alexander et al.
  - *templates for universal problems*
  - *Quiet backs*
- Software Researchers are applying design patterns as means of providing generic building blocks with which software can be constructed.
  - *JAVA, API and JDK*

Design Patterns: In software engineering

- A **general reusable solution** to a commonly occurring problem within a given context in software design.
- A design pattern is **not a finished design** that can be transformed directly into source or machine code. It is a **description or template for how to solve a problem that can be used in many different situations**. Patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system.
- **Object-oriented design patterns** typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved.

Pareto principle

- The **Pareto principle** (also known as the **80–20 rule** and the **law of the vital few**) states that, for many events, roughly 80% of the effects come from 20% of the causes.
- in 1896, published his first paper "Cours d'économie politique." Essentially, Pareto showed that approximately **80% of the land in Italy was owned by 20% of the population**; Pareto developed the principle by observing that 20% of the pea pods in his garden contained 80% of the peas.
- It is a common rule of thumb in business; e.g., "**80% of your sales come from 20% of your clients”**.

Pareto principle: In Software Engineering

- In computer science and engineering control theory, such as for **electromechanical energy converters**, the Pareto principle can be applied to **optimization** efforts:
- For example, Microsoft noted that by fixing the top 20% of the most-reported bugs, 80% of the related errors and crashes in a given system would be eliminated.
- In software engineering, Lowell Arthur expressed a corollary principle: "**20 percent of the code has 80 percent of the errors. Find them, fix them**!"

[Reference from Wikipedia]

Testing

- Complete testing impossible.
- Pareto principle:
  - small number of modules within a large software system is more problematic than the rest.
- Glass-Box Testing:
  - software interior visible to tester
  - Basis Path Testing
    - Every line must be executed at least once.
- Black-Box Testing: User point of view
  - Boundary Value Analysis: Extreme ranges and demanded activities
  - Applying redundancy: two software applying same task.
  - Shrink wrapped: beta testing
    - Benefits: feedback & marketing

Documentation:

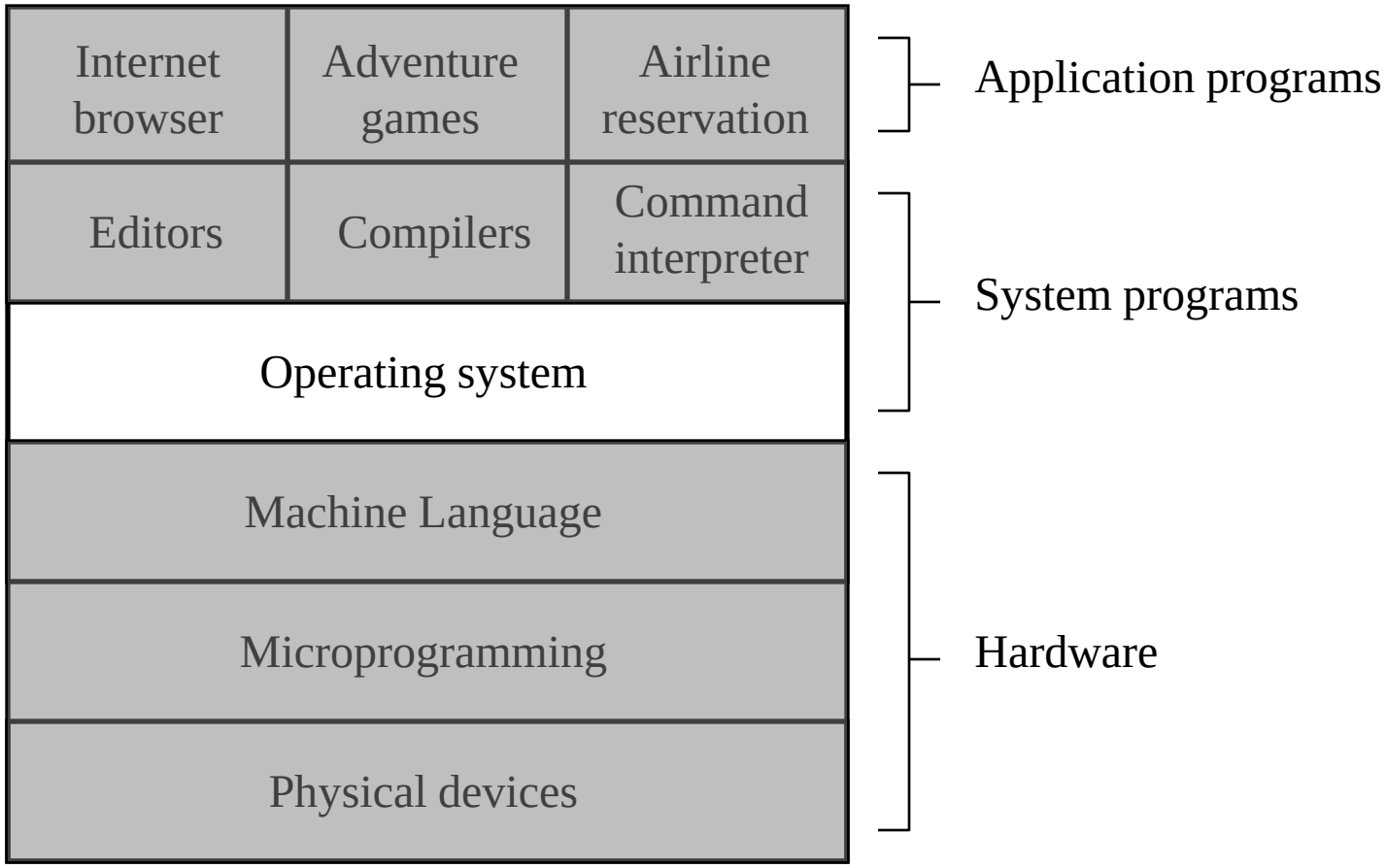
- User Documents
- Good documentation
  - books and technical writers
- Manual and Packages
- Internal composition of Software
- High level language
  - Comments
  - Indentions
  - Conventions
- Design Documents
- Updating by CASE

CHAPTER 3

Operating Systems (and Networks)

- Using hardware directly is highly complicated
  - even at the machine language level
- Especially so when multiple users want to perform multiple tasks - all at the same time
  - abstraction layer: *Operating System*

3.X: Computer System Overview

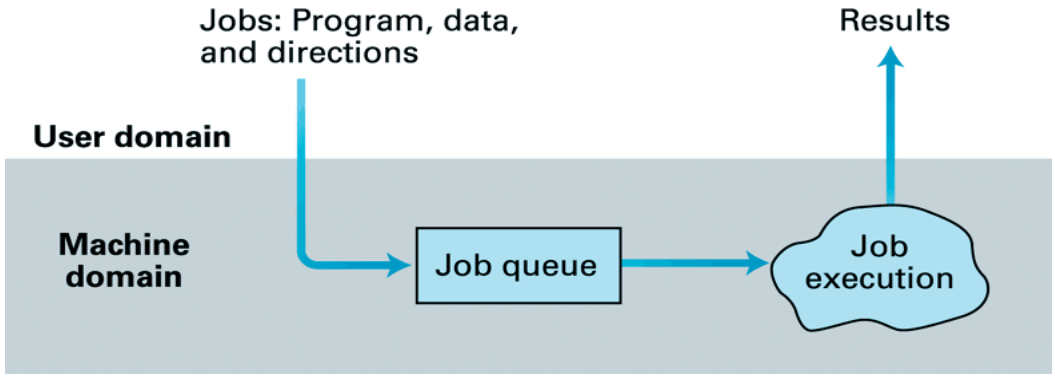


3.X: What is an Operating System?

- Top-down view: operating system is there to present the user with the equivalent of a '*virtual machine*'
  - hardware is difficult to program
  - user should not be annoyed with low level details
  - OS =>high-level abstractions (files, device access,.)
- Bottom-up view: operating system is there to manage all the pieces of a complex system
  - orderly, controlled management of *multiple programs* running at the same time
  - if needed: orderly, controlled management of *multiple users* at the same time

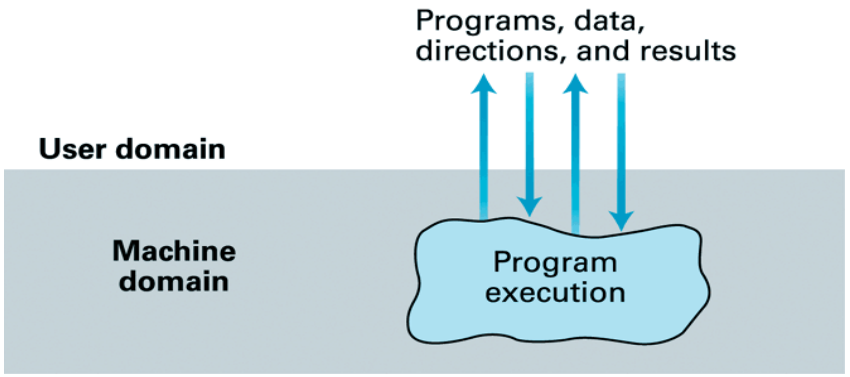
3.1: The Evolution of Operating Systems (1)

- 1945-1955:
  - User/programmer was ‘operating system’
- 1955-1965:
  - Human operator was ‘operating system’
  - Advent of *‘batch processing’*:



3.1: The Evolution of Operating Systems (2)

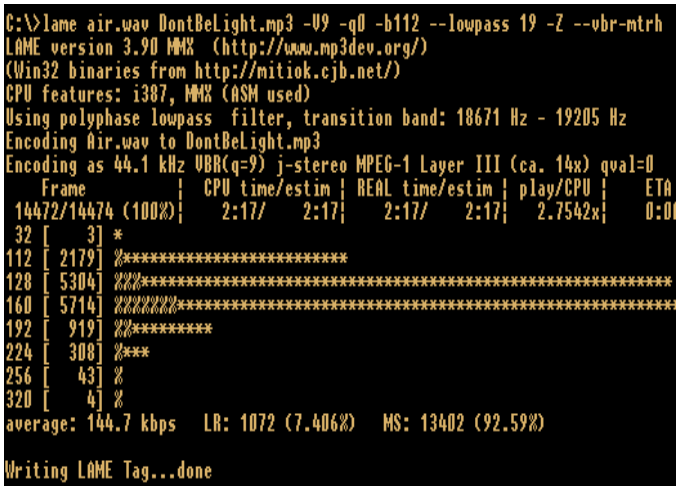
- 1965-1980:
  - Advent of *'interactive processing'*:



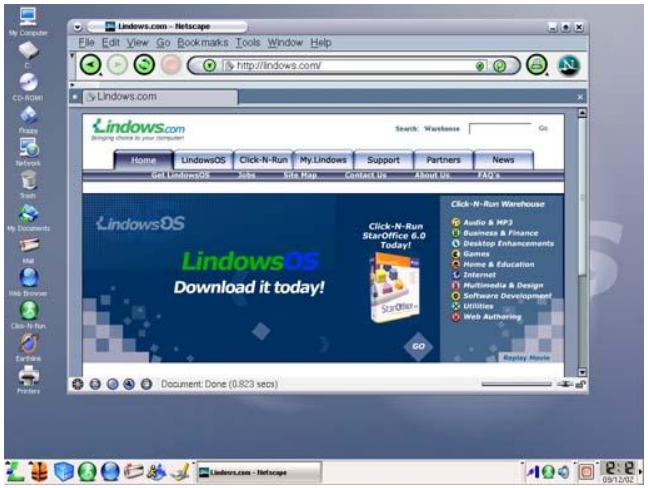
  - Provide services in a timely manner
    - *'real-time processing'*
  - Multitasking (single-user) & time-sharing (multi-user)

3.1: The Evolution of Operating Systems (3)

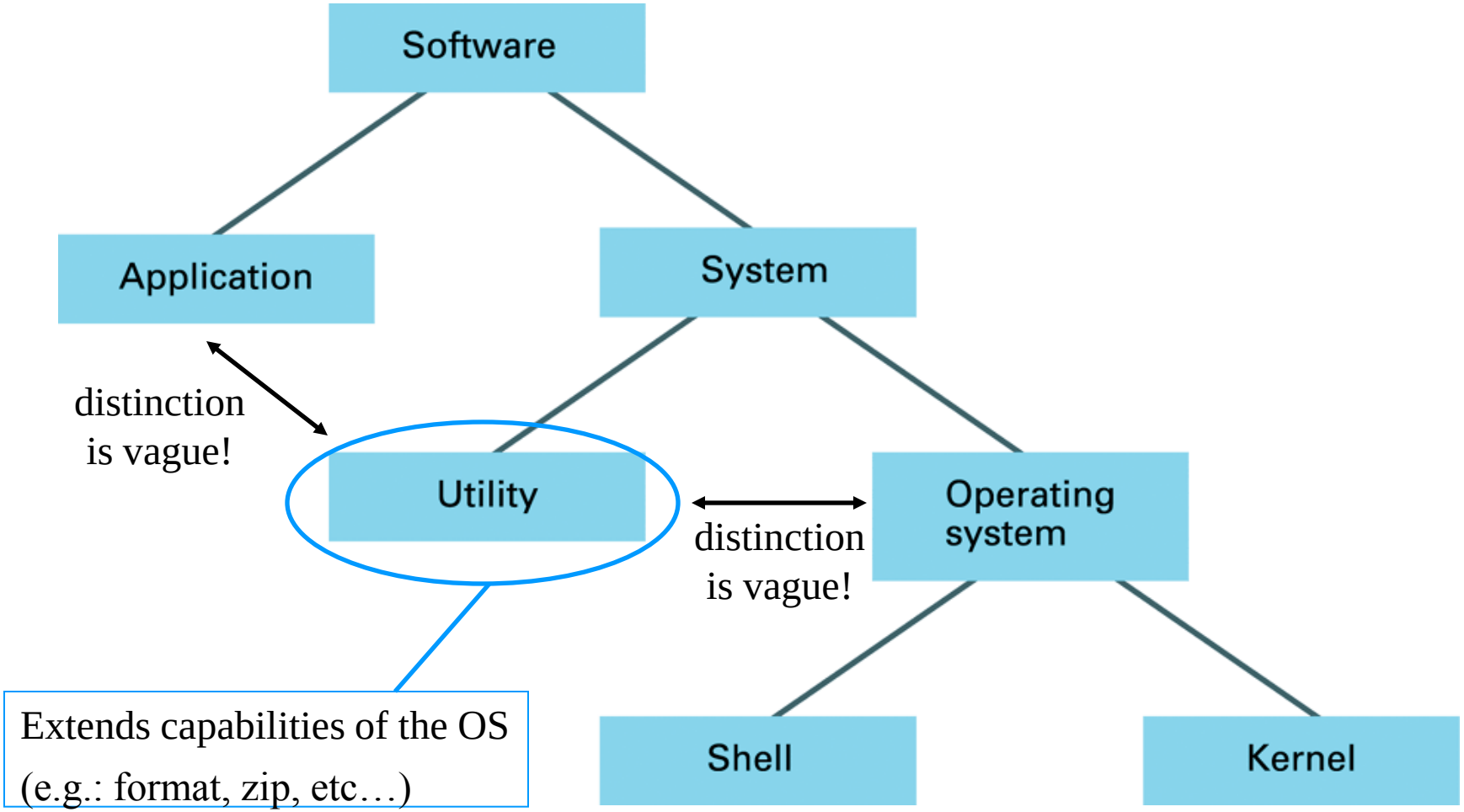
- 1980-now:
  - Operating systems for multi-processor architectures
    - includes: *load balancing*
  - Focus on user-friendliness:
    - especially: Graphical User Interface (GUI)



=>



3.2: Software classification



3.2: Components of an Operating System

- Interface between the OS and users:
  - shell (command-line, or GUI incl. window manager)
- Internal part of OS:
  - kernel :
    - file manager (coordinates the use of mass storage)
      - {Directory, folder, path and file descriptor}
    - memory manager (coordinates the use of main memory, especially in single-user or multi-user environments)
      - {Virtual Memory: Pages created and stored on Mass storage}
    - device drivers (for communication with external device controllers)
    - scheduler (coordinates the execution of multiple activities)
    - dispatcher (controls the allocation of time slices to activities)

3.3: The Concept of a Process (1)

- Important is distinction between a ‘program’ and the ‘activity of executing a program’!
  - Program is a *static* set of directions
  - Activity is *dynamic*, and its properties may change over time **=>** ‘*process*’
- Process has state, including:
  - current position in program (value of the program counter)
  - values in general-purpose registers & memory cells
- So: state is snapshot of machine at certain time

3.3: The Concept of a Process (2)

- A single program may run multiple processes
  - Example: multi-user word-processing program
    - two users edit separate documents at same time
    - both use the same copy of the program in main memory, but each with its specific set of data & process states
- In a computer system many processes compete for *time slices*
- Coordination / administration of these is one of most important tasks of time-sharing OS…

3.3: Process Administration & Time-sharing

- Process administration handled by
  - (1) scheduler
    - keeps track of all processes by maintaining a *process table*
      - *Entries like memory area and Priority (wait or ready)*
  - (2) dispatcher
    - ensures that scheduled processes are executed by dividing time into slices/ quantum , and switching CPU's attention among the processes
      - *Interrupt handler*