Software Engineering

Organization of this Lecture:

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

What is Software Engineering?

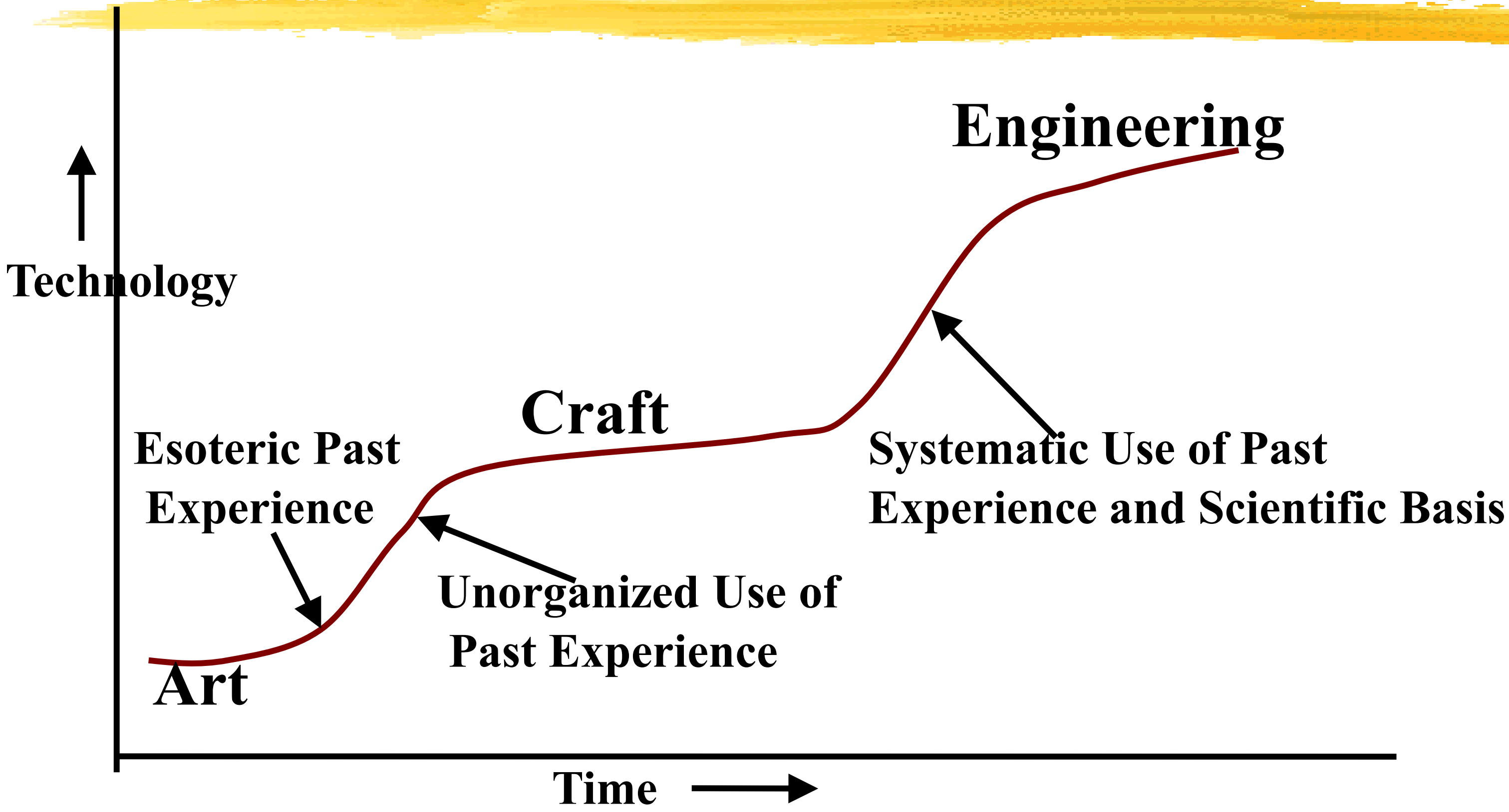
- Engineering approach to develop software.
  - Building Construction Analogy.

- Systematic collection of past experience:
  - techniques,
  - methodologies,
  - guidelines.

Engineering Practice

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

Technology Development Pattern

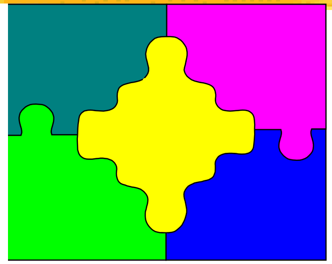


Why Study Software Engineering? (1)

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

Why Study Software Engineering? (2)

- Ability to solve complex programming problems:
  - How to break large projects into smaller and manageable parts?



- Learn techniques of:
  - specification, design, interface development, testing, project management, etc.

Why Study Software Engineering? (3)

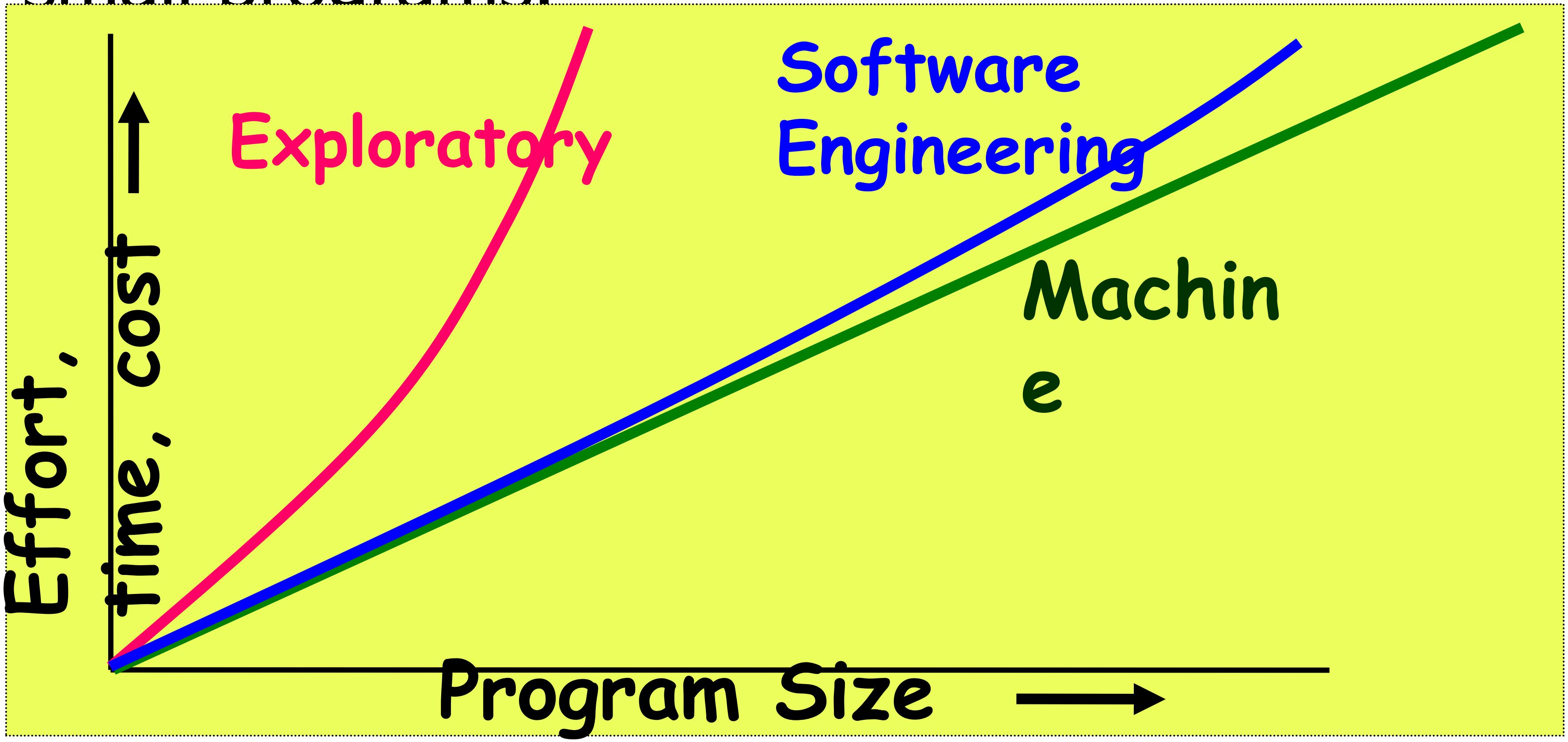
- To acquire skills to be a better programmer:
  - Higher Productivity
  - Better Quality Programs

Exploratory Development

- Early programmers used an exploratory (also called build and fix) style.
  - A `dirty' program is quickly developed.
  - The bugs are fixed as and when they are noticed.
  - Similar to a first year student develops programs.

What is Wrong with the Exploratory Style?

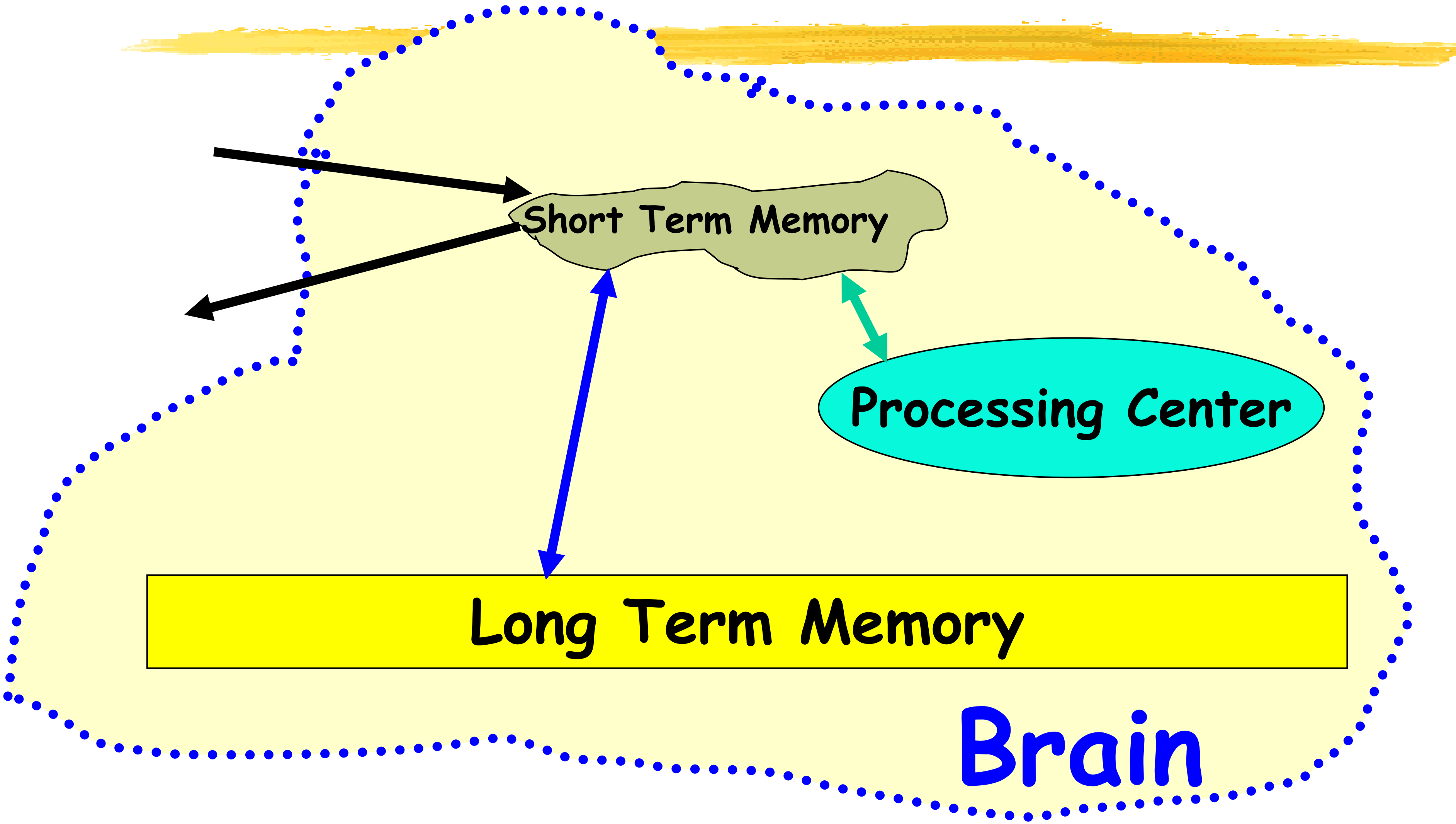
- Can successfully be used for developing rather small programs.



An Interpretation Based on Human Cognition Mechanism

- Human memory can be thought to be made up of two distinct parts [Miller 56]:
  - Short term memory and
  - Long term memory.

Schematic Representation of Brain



Short Term Memory

- An item stored in the short term memory can get lost:
  - Either due to decay with time or
  - Displacement by newer information.
- This restricts the time for which an item is stored in short term memory:
  - To few tens of seconds.
  - However, an item can be retained longer in the short term memory by recycling.

Evidence of Short Term Memory

- Short term memory is evident:
  - In many of our day-to-day experiences.
- Suppose, you look up a number from the telephone directory and start dialling it.
  - If you find the number is busy, you can dial the number again after a few seconds without having to look up the directory.
- But, after several days:
  - You may not remember the number at all
  - Would need to consult the directory again.

The Magical Number 7

- If a person deals with seven or less number of items:
  - These would be easily be accommodated in the short term memory.
  - So, he can easily understand it.
- As the number of new information increases beyond seven:
  - It becomes exceedingly difficult to understand it.

Implication in Program Development

- A small program having just a few variables:
  - Is within easy grasp of an individual.
- As the number of independent variables in the program increases:
  - It quickly exceeds the grasping power of an individual:
    - Requires an unduly large effort to master the problem.

Implication in Program Development

- Instead of a human, if a machine could be writing (generating) a program,
  - The slope of the curve would be linear.
- But, why does the effort-size curve become almost linear when software engineering principles are deployed?
  - Software engineering principles extensively use techniques specifically targeted to overcome the human cognitive limitations.

Software Engineering to Overcome Human Cognitive Limitations

- Two important principles are profusely used:
  - Abstraction
  - Decomposition

What is Abstraction?

- Simplify a problem by omitting unnecessary details.
  - Focus attention on only one aspect of the problem and ignore irrelevant details.
  - Also called model building.

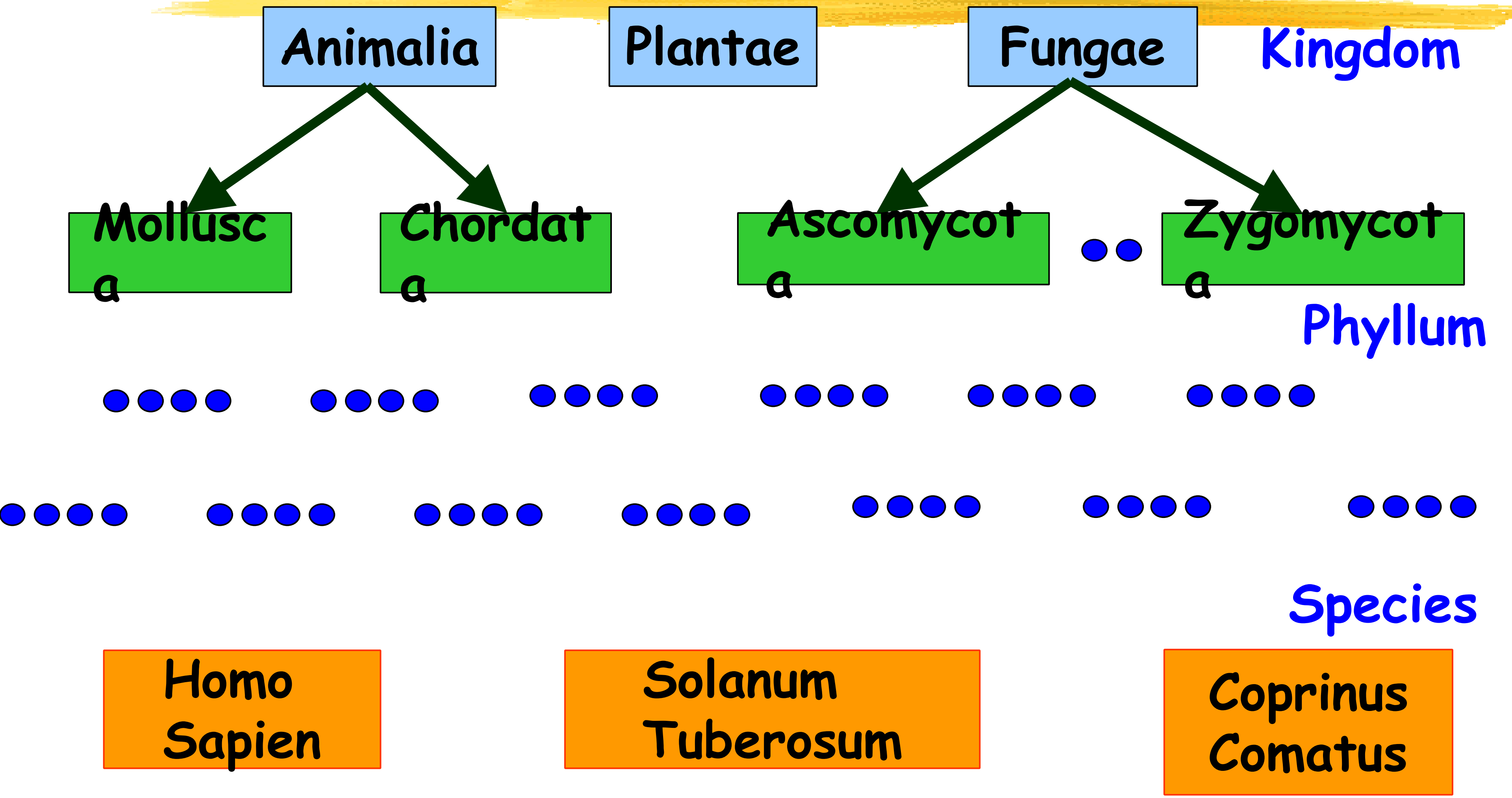
Abstraction

- For complex problems:
  - A single level of abstraction is inadequate.
  - A hierarchy of abstractions needs to be constructed.
- Hierarchy of models:
  - A model in one layer is an an abstraction of the lower layer model.
  - An implementation of the model at the higher layer.

Abstraction Example

- Suppose you are asked to understand the life forms that inhabit the earth.
- If you start examining each living organism:
  - You will almost never complete it.
  - Also, get thoroughly confused.
- You can build an abstraction hierarchy.

Living Organisms



Decomposition

- Decompose a problem into many small independent parts.
  - The small parts are then taken up one by one and solved separately.
  - The idea is that each small part would be easy to grasp and can be easily solved.
  - The full problem is solved when all the parts are solved.

Decomposition

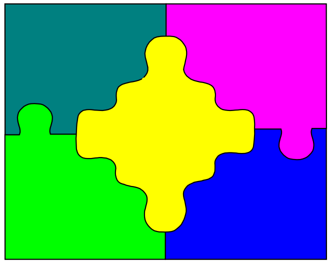
- A popular use of decomposition principle:
  - Try to break a bunch of sticks tied together versus breaking them individually.
- Any arbitrary decomposition of a problem may not help.
  - The decomposed parts must be more or less independent of each other.

Decomposition Example

- Example use of decomposition principle:
  - You understand a book better when the contents are organized into independent chapters.
  - Compared to when everything is mixed up.

Why Study Software Engineering? (Coming back)

- To acquire skills to develop large programs.
  - Handling exponential growth in complexity with size.
  - Systematic techniques based on abstraction (modelling) and decomposition.



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.

Programs versus Software Products

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

Emergence of Software Engineering

- <u>Early Computer Programming (1950s):</u>
  - Programs were being written in assembly language.
  - Programs were limited to about a few hundreds of lines of assembly code.

Early Computer Programming (50s)

- Every programmer developed his own style of writing programs:
  - according to his intuition (exploratory programming).

High-Level Language Programming (Early 60s)

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

High-Level Language Programming (Early 60s)

- Software development style was still exploratory.
  - Typical program sizes were limited to a few thousands of lines of source code.

Control Flow-Based Design
(late 60s)

- Size and complexity of programs increased further:
  - exploratory programming style proved to be insufficient.
- Programmers found:
  - very difficult to write cost-effective and correct programs.

Control Flow-Based Design
(late 60s)

- Programmers found:
  - programs written by others very difficult to understand and maintain.
- To cope up with this problem, experienced programmers advised: ``Pay particular attention to the design of the program's control structure.''

Control Flow-Based Design (late 60s)

- A program's control structure indicates:
  - the sequence in which the program's instructions are executed.
- To help design programs having good control structure:
  - flow charting technique was developed.

Control Flow-Based Design (late 60s)

- Using flow charting technique:
  - one can represent and design a program's control structure.
  - Usually one understands a program:
    - by mentally simulating the program's execution sequence.

Control Flow-Based Design

(Late 60s)

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

Control Flow-Based Design (Late 60s)

- It was found:
  - GO TO statements makes control structure of a program messy
  - GO TO statements alter the flow of control arbitrarily.
  - The need to restrict use of GO TO statements was recognized.

Control Flow-Based Design (Late 60s)

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

Control-flow Based Design (Late 60s)

- At that time, Dijkstra published his article:
  - "Goto Statement Considered Harmful" Comm. of ACM, 1969.
- Many programmers were unhappy to read his article.

Control Flow-Based Design (Late 60s)

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

Control Flow-Based Design (Late 60s)

- But, soon it was conclusively proved:
  - only three programming constructs are sufficient to express any programming logic:
    - sequence (e.g. a=0;b=5;)
    - selection (e.g.if(c=true) k=5 else m=5;)
    - iteration (e.g. while(k>0) k=j-k;)

Control-flow Based Design (Late 60s)

- Everyone accepted:
  - it is possible to solve any programming problem without using GO TO statements.
  - This formed the basis of Structured Programming methodology.

Structured Programming

- A program is called structured
  - when it uses only the following types of constructs:
    - sequence,
    - selection,
    - iteration

Structured programs

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

Structured Programming

- Research experience shows:
  - programmers commit less number of errors
    - while using structured if-then-else and do-while statements
    - compared to test-and-branch constructs.

Data Structure-Oriented Design (Early 70s)

- Soon it was discovered:
  - it is important to pay more attention to the design of data structures of a program
    - than to the design of its control structure.

Data Structure-Oriented Design (Early 70s)

- Techniques which emphasize designing the data structure:
  - derive program structure from it:
    - are called `data structure-oriented design techniques.`

Data Structure Oriented Design (Early 70s)

- Example of data structure-oriented design technique:
  - Jackson's Structured Programming(JSP) methodology
    - Developed in 1970s.

Data Structure Oriented Design (Early 70s)

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

Data Structure Oriented Design (Early 70s)

- In JSP methodology:
  - a program's data structures are first designed using notations for
    - sequence, selection, and iteration.
  - Then data structure design is used :
    - to derive the program structure.

Data Structure Oriented Design (Early 70s)

- Several other data structure-oriented Methodologies also exist:
  - e.g., Warnier-Orr Methodology.

Data Flow-Oriented Design (Late 70s)

- Data flow-oriented techniques advocate:
  - the data items input to a system must first be identified,
  - processing required on the data items to produce the required outputs should be determined.

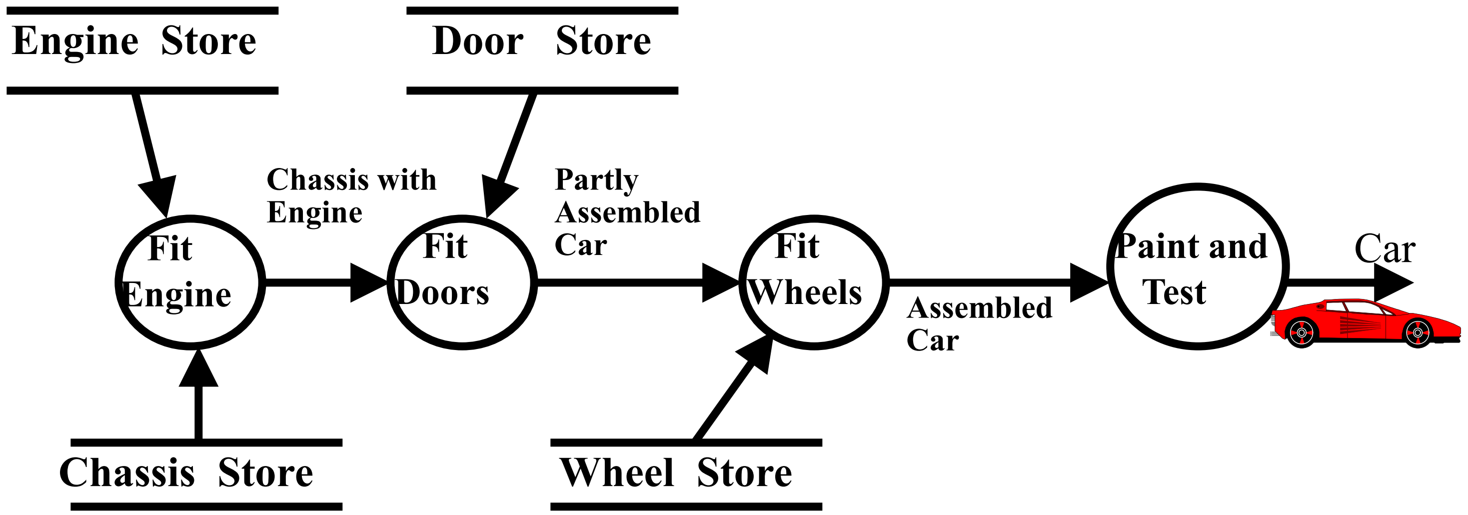
Data Flow-Oriented Design (Late 70s)

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

Data Flow-Oriented Design (Late 70s)

- Data flow technique is a generic technique:
  - can be used to model the working of any system
    - not just software systems.
- A major advantage of the data flow technique is its simplicity.

Data Flow Model of a Car Assembly Unit



Object-Oriented Design (80s)

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

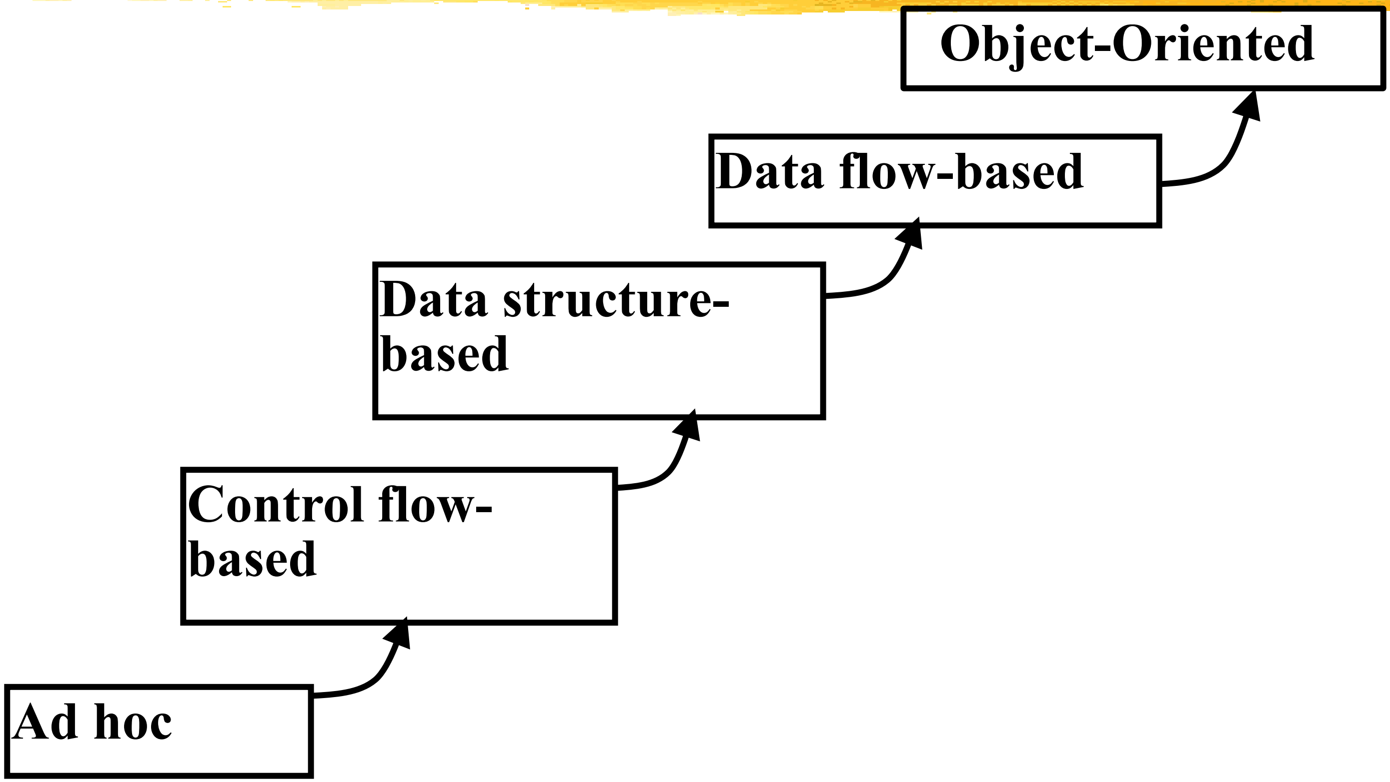
Object-Oriented Design (80s)

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

Object-Oriented Design (80s)

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

Evolution of Design Techniques



Evolution of Other Software Engineering Techniques

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

Evolution of Other Software Engineering Techniques

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

Differences between the exploratory style and modern software development practices

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

Differences between the exploratory style and modern software development practices

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

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

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

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

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

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

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

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

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

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

- There is better visibility of design and code:
  - visibility means production of good quality, consistent and standard documents.
  - In the past, very little attention was being given to producing good quality and consistent documents.
  - We will see later that increased visibility makes software project management easier.

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

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

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

- Projects are being thoroughly planned:
  - estimation,
  - scheduling,
  - monitoring mechanisms.

Software Life Cycle

- Software life cycle (or software process):
  - series of identifiable stages that a software product undergoes during its life time:
    - Feasibility study
    - requirements analysis and specification,
    - design,
    - coding,
    - testing
    - maintenance.

Life Cycle Model

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

Life Cycle Model (CONT.)

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

Why Model Life Cycle ?

- A written description:
  - forms a common understanding of activities among the software developers.
  - helps in identifying inconsistencies, redundancies, and omissions in the development process.
  - Helps in tailoring a process model for specific projects.

Life Cycle Model (CONT.)

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

Life Cycle Model (CONT.)

- A life cycle model:
  - defines entry and exit criteria for every phase.
  - A phase is considered to be complete:
    - only when all its exit criteria are satisfied.

Life Cycle Model (CONT.)

- It becomes easier for software project managers:
  - to monitor the progress of the project.

Life Cycle Model (CONT.)

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

Summary

- Software engineering is:
  - systematic collection of decades of programming experience
  - together with the innovations made by researchers.

Summary

- A fundamental necessity while developing any large software product:
  - adoption of a life cycle model.

Summary

- Adherence to a software life cycle model:
  - helps to do various development activities in a systematic and disciplined manner.
  - also makes it easier to manage a software development effort.

Reference

- R. Mall, "Fundamentals of Software Engineering," Prentice-Hall of India, 2011, <u>CHAPTER 1.</u>