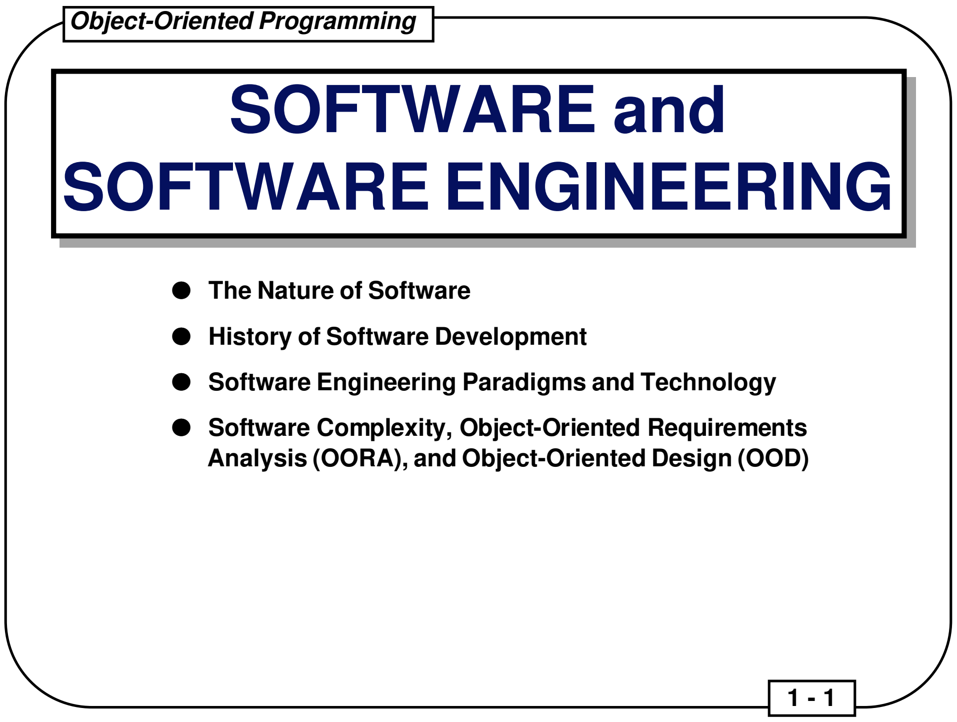

Object-Oriented Programming

SOFTWARE and SOFTWARE ENGINEERING

- The Nature of Software
- History of Software Development
- Software Engineering Paradigms and Technology
- Software Complexity, Object-Oriented Requirements Analysis (OORA), and Object-Oriented Design (OOD)

1 - 1

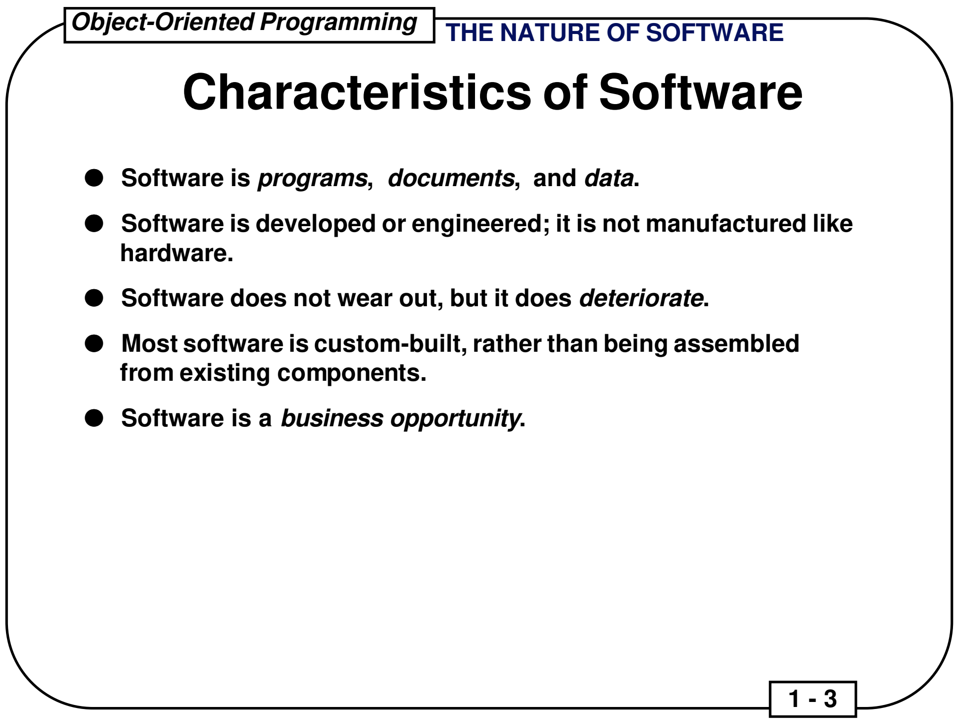
Objectives of Module 1

- Present and discuss the idea that software is much more than just code -- *engineered* software is composed of *controlled configuration items* which include documents, data, and code
- Present and discuss the history of software development, including its evolution into a business
- Present and discuss several different software engineering paradigms, showing different methods for developing engineered software:
  - ❍ Classic "waterfall" method
  - ❍ Rapid prototyping
  - ❍ Spiral method
- Introduce the concepts of complexity, Object-Oriented Requirements Analysis (OORA), and Object-Oriented Design (OOD)

THE NATURE OF SOFTWARE

- ✓ Characteristics of Software
- ✓ Failure Curves for Hardware and Software
- ✓ Software Components
- ✓ Software Configuration

- **The Nature of Software**
- *History of Software Development*
- *Software Engineering Paradigms and Technology*
- *Software Complexity, Object-Oriented Requirements Analysis (OORA), and Object-Oriented Design (OOD)*


Object-Oriented Programming THE NATURE OF SOFTWARE

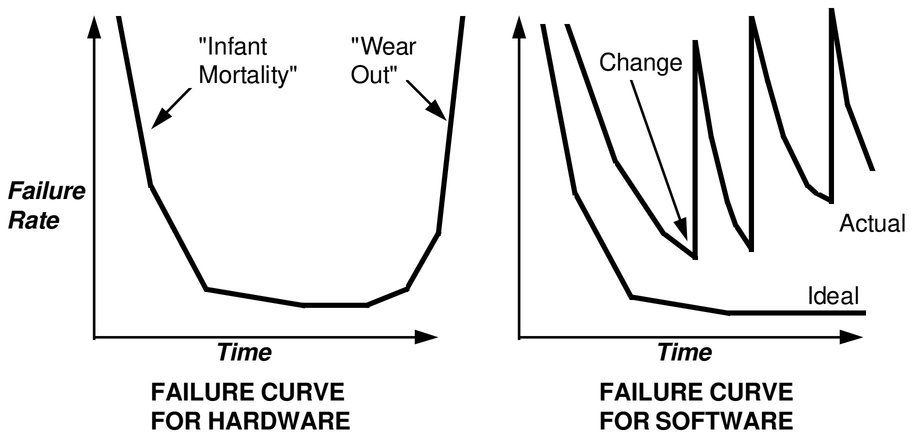
Characteristics of Software

- **Software is *programs*, *documents*, and *data*.**
- **Software is developed or engineered; it is not manufactured like hardware.**
- **Software does not wear out, but it does *deteriorate*.**
- **Most software is custom-built, rather than being assembled from existing components.**
- **Software is a *business opportunity*.**

1 - 3

1. Many people have the non-engineering view of software:
   - as computer programs (*i.e.*, source code and/or executables),
   - as data structures (*e.g.*, data base schemas), and
   - as operation and user documentation (usually created as an afterthought when the "real" work is done)
2. A major factor in the speed at which quality software is developed is the failure to reuse software:
   - there are few reusable component libraries,
   - there is a bias against using "old" routines or routines "not invented here", and
   - software as a creative art is a perception that is held by many people.
3. Software has become a *business opportunity*, where the success or failure of a business (and the jobs of the people associated with it) depends upon the timely development of quality software. The customer is demanding both high quality in the software (and once a business has a reputation of putting out "junk", word gets around quickly) and timeliness of delivery (the customer wants the software *now*).

Failure Curves for Hardware and Software

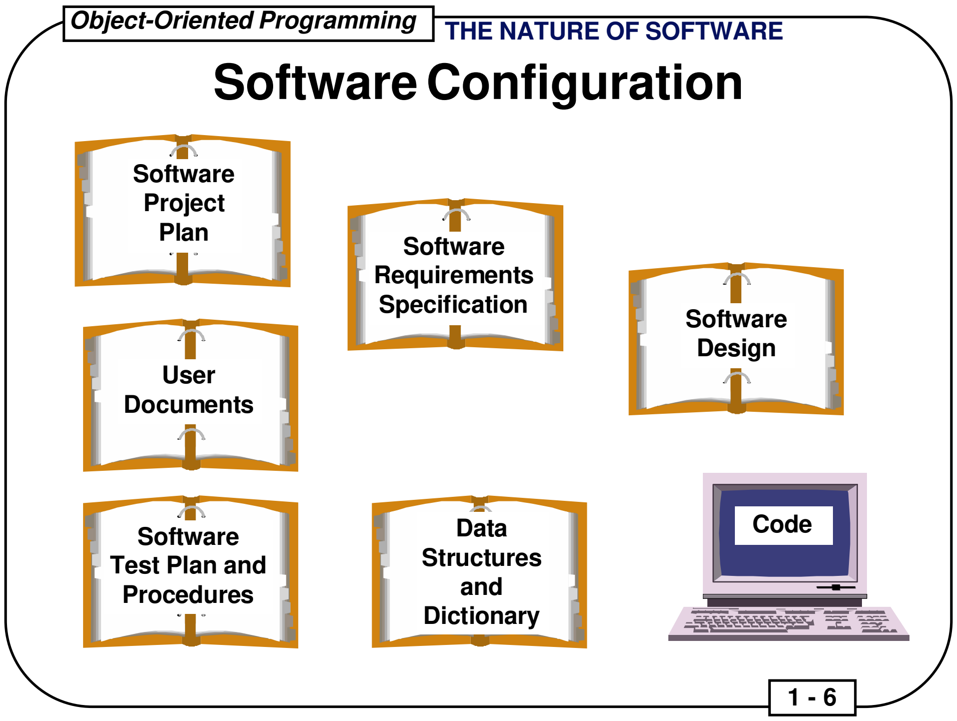


Hardware tends to have a wear-in time during which it has a higher probability of failure. This is generally referred to as *infant mortality*. Once the initial period is passed, hardware tends to operate without failure until components age enough to cause breakdown.

Software also shows an early error rate, but updates should remove the most obvious problems which render the software unreliable. Updates for added functionality often add more errors, as is shown by the spikes on the failure curve for software. As updates are made, more latent errors appear in the software to make it inherently less reliable until the software is finally considered unreliable enough to stop using the software product or to perform a major redesign and rewrite of the software.

Software Components

- Software programs, or software systems, consist of *components*.
- A set of components which comprise a logical unit of software is called a *software configuration item*.
- Reuse and development of reliable, trusted software components improves software *quality* and *productivity*.
- Computer language forms:
  - Machine level (microcode, digital signal generators)
  - Assembly language (PC assembler, controllers)
  - High-order languages (FORTRAN, Pascal, C, Ada, ...)
  - Specialized languages (LISP, OPS5, Prolog, ...)
  - Fourth generation languages (databases, windows apps)


Object-Oriented Programming

THE NATURE OF SOFTWARE

Software Configuration

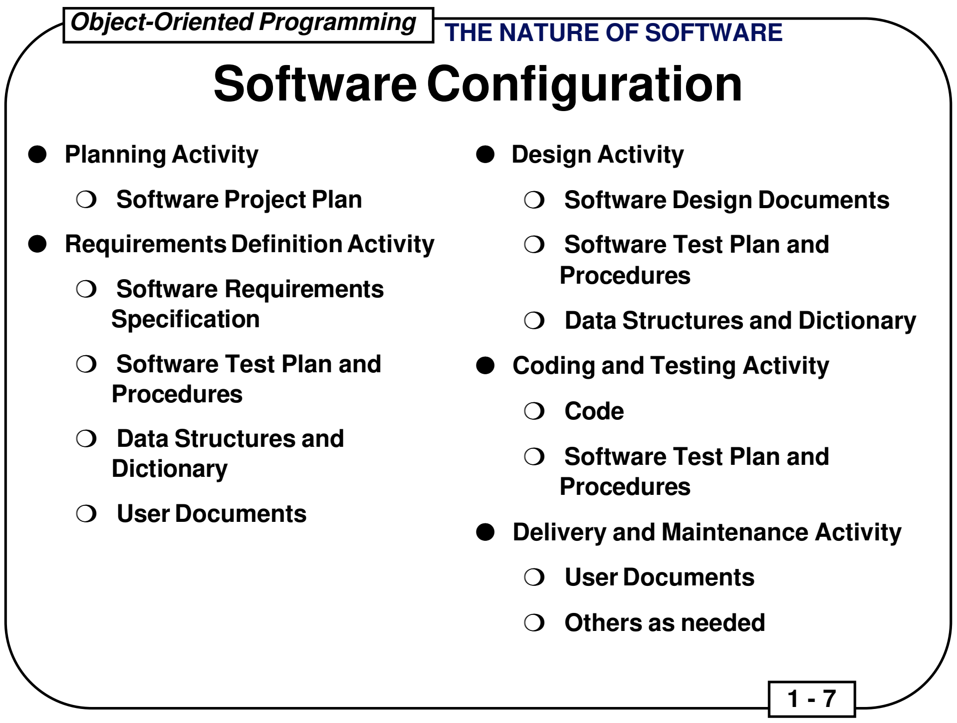
Software Project Plan
Software Requirements Specification
Software Design
User Documents
Software Test Plan and Procedures
Data Structures and Dictionary
Code

1 - 6

Composition of Software

The software we develop is composed of these parts, also known as *software configuration items*:

- **Software Project Plan** - A document which details the tasks, schedules, needed resources, and approach to carry out development. This is the first document produced and it includes cost details.
- **Software Requirements Specification** - A document which identifies *what* is required of the software (as opposed to the design document, which describes *how* to implement the software). This document includes information on how implementation of the requirements will be verified (*i.e.*, some initial test considerations). This very important document is often quite time consuming to produce.
- **Software Test Plan and Procedures** - A document which describes the test methods, approaches, procedures, and the support required for testing the software code components and the integrated software system. This document includes test data and expected results and is developed during both the requirements definition and design phases of the project.
- **Data Structures and Dictionary** - The Data Dictionary documents all data structures and the definitions of terms, variables, and other items of interest regarding the details of the data in the system. It supports software design, coding, and maintenance and is developed during the requirements and design phases.
- **Software Design Document** - A document which clearly details the behavior and structure of the system as a whole and each software code component.
- **User Documents** - These are user guides, reference guides, application notes, and other items deemed necessary for the users.
- **Code** - The compilable source code of the system.

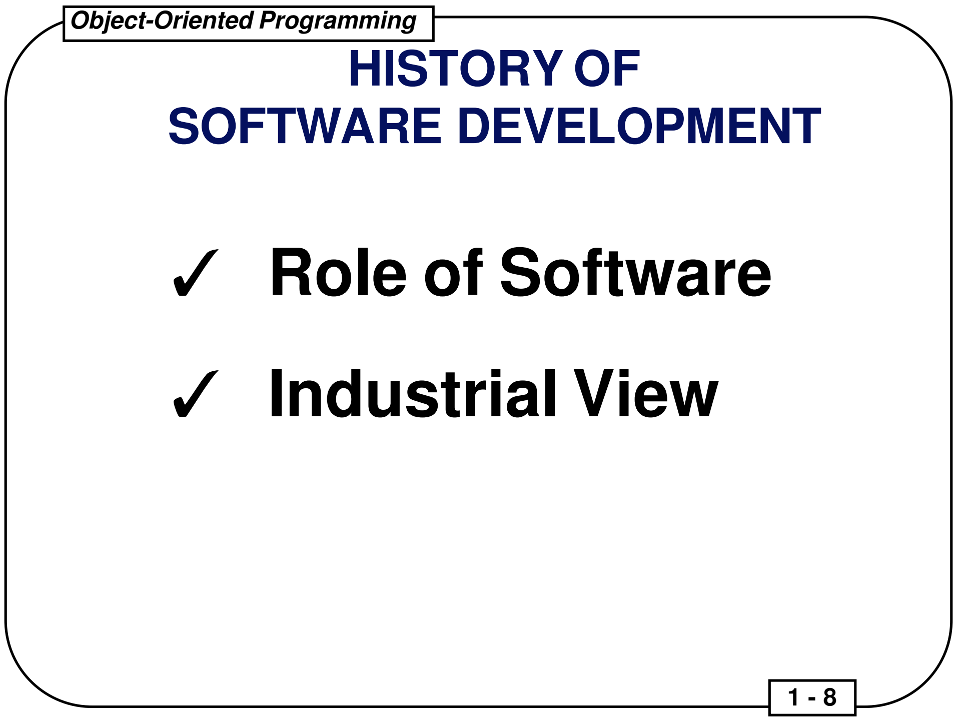


Software Configuration

- Planning Activity
  - ❍ Software Project Plan
- Requirements Definition Activity
  - ❍ Software Requirements Specification
  - ❍ Software Test Plan and Procedures
  - ❍ Data Structures and Dictionary
  - ❍ User Documents
- Design Activity
  - ❍ Software Design Documents
  - ❍ Software Test Plan and Procedures
  - ❍ Data Structures and Dictionary
- Coding and Testing Activity
  - ❍ Code
  - ❍ Software Test Plan and Procedures
- Delivery and Maintenance Activity
  - ❍ User Documents
  - ❍ Others as needed

When are the Software Configuration Items Produced?

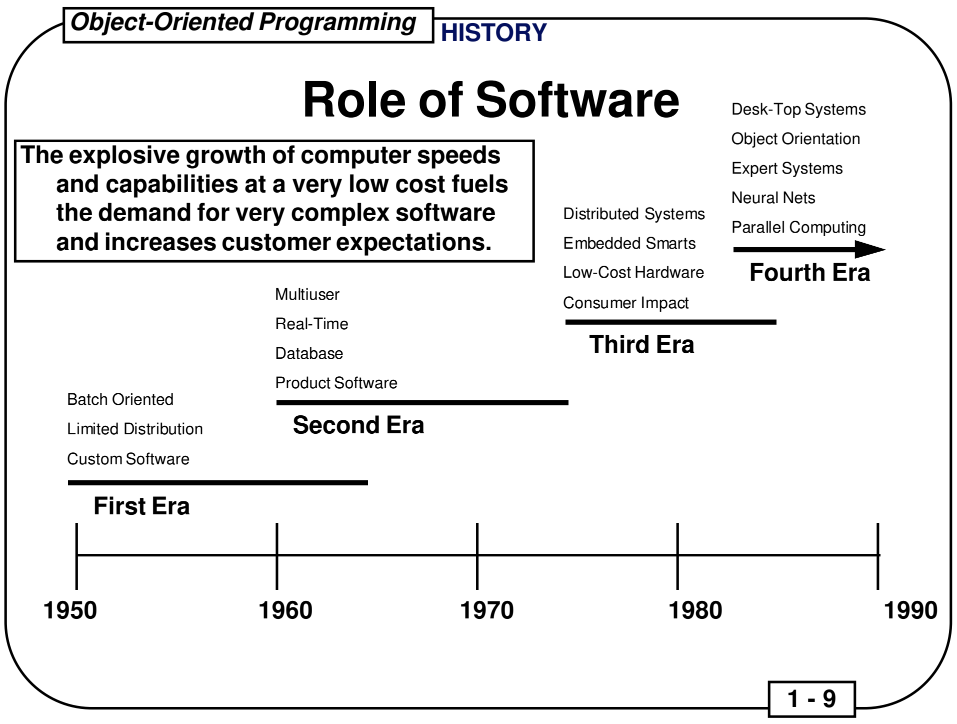
- The *Software Configuration Items* are drafted, reviewed, revised, etc., at many points throughout the activities performed during the development of the software. Seldom is a Software Configuration Item felt to be completely finished.
- All *Software Configuration Items* are placed under *configuration control*, allowing for them to be changed and all changes to them to be tracked. Any particular version of any of the configuration items may be recreated when desired.
- The control of the Software Configuration Items extends from the planning stages of the project through the maintenance activities -- the entire life of the software.



HISTORY OF SOFTWARE DEVELOPMENT

- ✓ **Role of Software**
- ✓ **Industrial View**

- ● *The Nature of Software*
- ● **History of Software Development**
- ● *Software Engineering Paradigms and Technology*
- ● *Software Complexity, Object-Oriented Requirements Analysis (OORA), and Object-Oriented Design (OOD)*



Role of Software

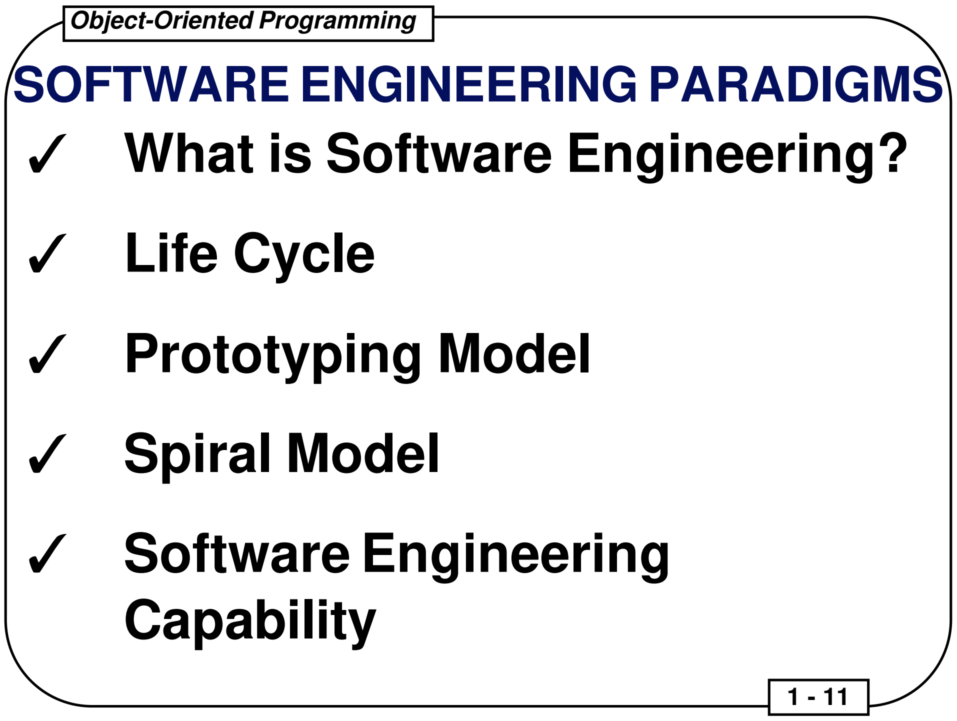
The explosive growth of computer speeds and capabilities at a very low cost fuels the demand for very complex software and increases customer expectations.

1. Early years (to about 1970):
   - large, expensive, few, protected computers
   - small programs inefficiently written
   - major constraints (memory, speed, I/O)
   - non-realtime batch-oriented software; single user
   - single programmer per program
2. Middle years (1970 to 1990):
   - realtime software development
   - multiple programmer teams
   - software development industry emerges
   - emerging interest in engineering the development of software
   - department-level computers make them more accessible; multiuser
3. Later years (1980 to 1990):
   - personal computer makes computing highly accessible
   - very large software industry develops
   - large programs and software systems emerge
   - hardware is distributed using networks
   - communications using computers evolves
   - software becomes highly departmentalized

Industrial View

- Why does it take so long to finish a working software system?
- Why are development costs so high?
- Why can't we find all software errors before software is delivered?
- How can we measure the progress of software development?
- How can we survive in the global economy?

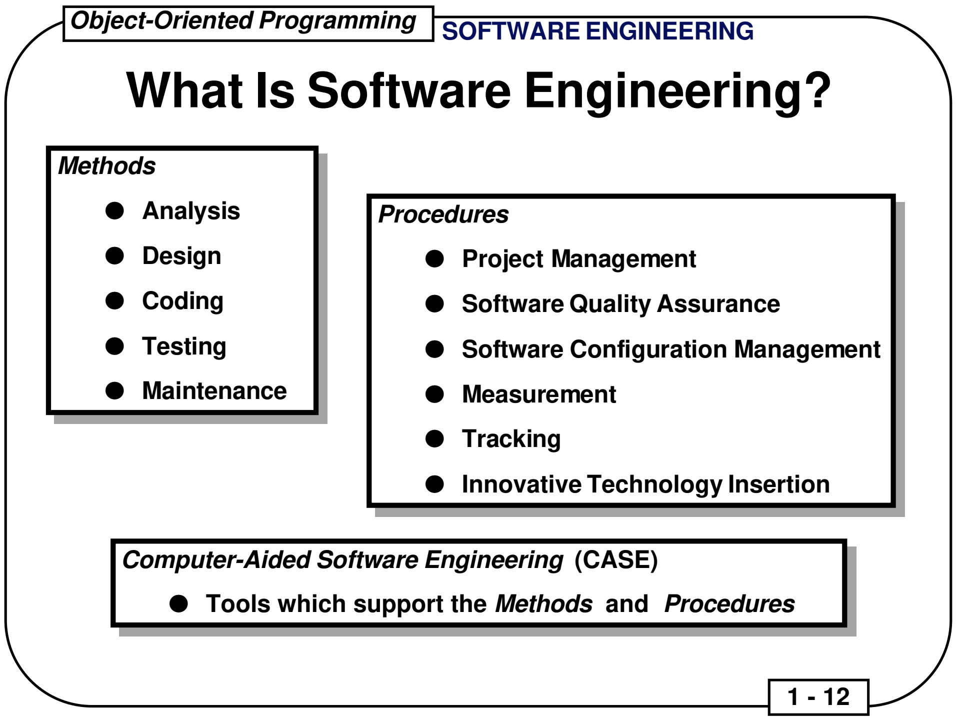
1. Early software development was considered to be an "art form"
2. Formal methods did not exist or were not followed
3. Programming education mainly by trial and error
4. Example of problems: Operating System for the IBM 360 (data extracted from **The Mythical Man-Month** by Fredrick Brooks, Addison-Wesley, 1975)
   - large software product (almost 1 million lines of code)
   - as errors were fixed, more errors were produced
   - adding people to the project made things worse
   - few formal methods of design were known or used
   - project was abandoned and the operating system was completely rewritten
   - project had a major impact on producing formal methods in software engineering



SOFTWARE ENGINEERING PARADIGMS

- ✓ **What is Software Engineering?**
- ✓ **Life Cycle**
- ✓ **Prototyping Model**
- ✓ **Spiral Model**
- ✓ **Software Engineering Capability**

- *The Nature of Software*
- *History of Software Development*
- **Software Engineering Paradigms and Technology**
- *Software Complexity, Object-Oriented Requirements Analysis (OORA), and Object-Oriented Design (OOD)*



What Is Software Engineering?

Methods

- **Analysis**
- **Design**
- **Coding**
- **Testing**
- **Maintenance**

Procedures

- **Project Management**
- **Software Quality Assurance**
- **Software Configuration Management**
- **Measurement**
- **Tracking**
- **Innovative Technology Insertion**

***Computer-Aided Software Engineering* (CASE)**

- **Tools which support the *Methods* and *Procedures***

The Essence of Software Engineering

Methods

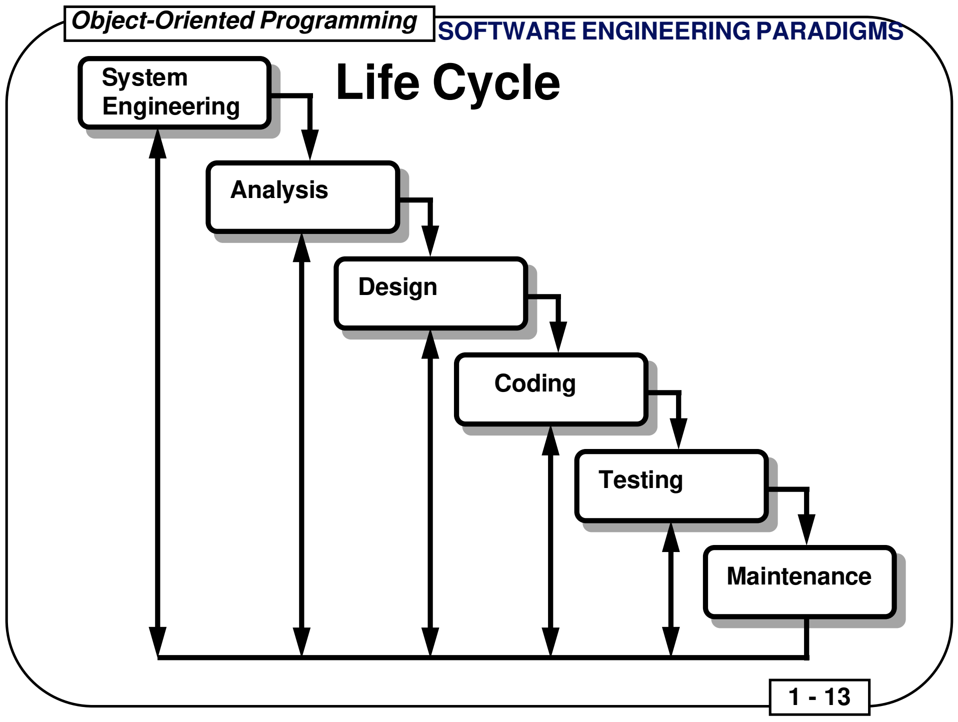
- *Methods* comprise the techniques used to perform the various phases of the software development
- *Methods* are not necessarily documented formally and are often unique to each organization and its culture
- Once a method is selected for a project, automated facilities may come into play to support the method; a common flaw in many organizations is that automated facilities are sometimes selected first and people then spend time figuring out how to apply the facility to their methods or adapt them methods to the facility

Procedures

- *Procedures* are formal, documented activities performed during the various phases of the software development
- Personnel with less advanced training are often employed in roles which implement the various procedures
- Implementation of the procedures is one of the best places to apply automated techniques

CASE Tools

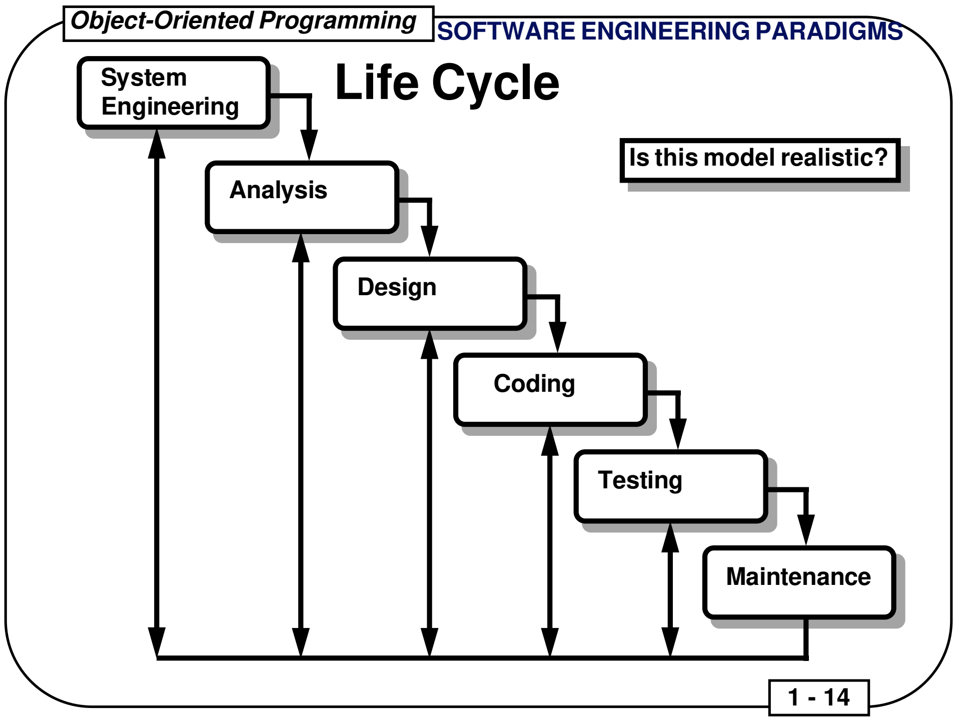
- *Computer-Aided Software Engineering tools* can be a valuable aid when applied to support a well-established method or set of methods
- CASE tools can also introduce a high degree of risk to a project if the organization is immature in its methods



Classic "Waterfall" Model

This model is a systematic, sequential approach to software development. It is the oldest and most often used of all software engineering paradigms.

- **System Engineering** - Establish requirements for the software as a part of the larger system. Determine which parts of the entire system are to be allocated to software.
- **Analysis** - Establish requirements from the point of view of the software. Include functional, performance, and interface requirements for the software subsystem.
- **Design** - Define the software architecture, procedural details, data structures, and interface characteristics for the software. The design process plans the implementation of the software to meet the requirements. Rapid prototyping and automated analysis of the design may come into play. The design of the software presents enough information so that a programmer who does not necessarily know how the system works can create code.
- **Coding** - The translation of a design into a compilable form. If the design is sufficiently detailed and adequate technologies are available, coding may be automatic.
- **Testing** - Analysis and verification that codes statements are fully compliant with the requirements and the customer's intent.
- **Maintenance** - The process of continuing to support the system after it is released to the customer. This process often involves several types of activities:
    - **Corrective Maintenance** - fixing errors
    - **Adaptive Maintenance** - changing the software to run in different environments (such as new versions of an OS or new target platforms)
    - **Enhancement** - adding new features to the software

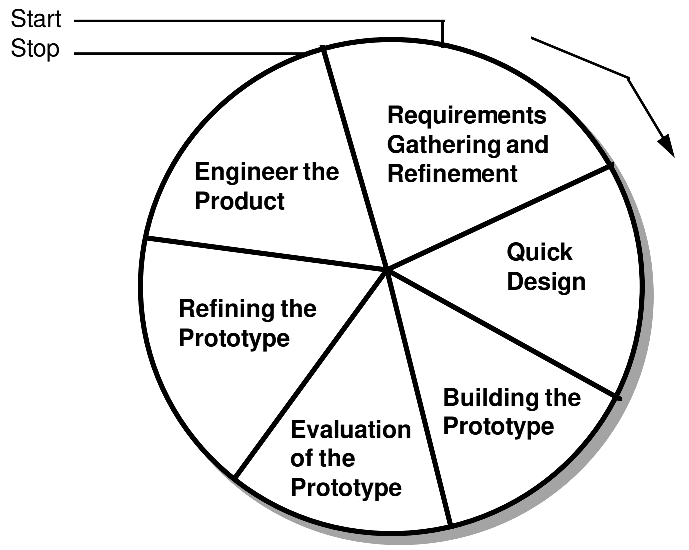


Life Cycle

Problems with the Classic "Waterfall" Model

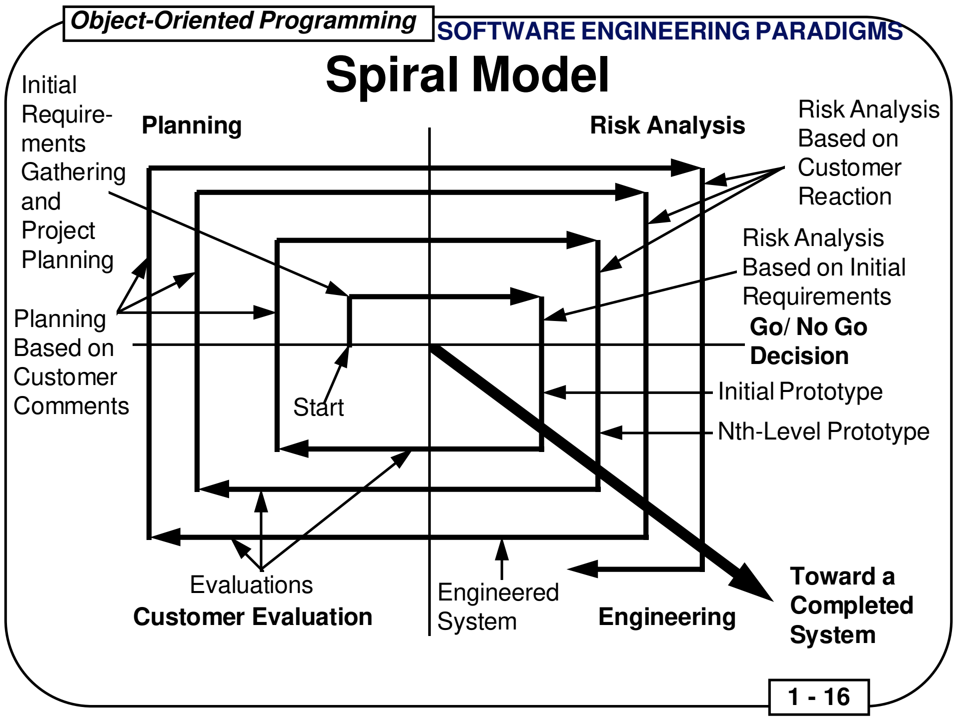
- Real projects rarely follow strict sequential development.
- It is very difficult to fully state all the requirements up front. The customer does not often know exactly what his requirements are or does not provide all the necessary input to fully state the requirements.
- This model demands patience from the customer. Working code is not available until very late into the project.

Prototyping Model



An Iterative Process

- **Requirements Gathering and Refinement** - During the first loop around this circle, an initial statement of the requirements is obtained. During later loops, the requirements statement is revised based on customer feedback.
- **Quick Design** - Very little time is usually spent on designing the prototype. Often, aided by workstation-based tools, we transition directly into building the prototype.
- **Building the Prototype** - This often involves the aid of software tools.
- **Evaluation of the Prototype** - The customer and the developers unite in their efforts to look at the prototype and determine its flaws.
- **Refining the Prototype** - This step is taken only if the prototype is not discarded.
- **Engineer the Product** - This step is taken when the customer and developer are completely satisfied.


Object-Oriented Programming
SOFTWARE ENGINEERING PARADIGMS
Spiral Model
Initial Require-ments Gathering and Project Planning
Planning
Risk Analysis
Risk Analysis Based on Customer Reaction
Risk Analysis Based on Initial Requirements
Go/ No Go Decision
Initial Prototype
Nth-Level Prototype
Planning Based on Customer Comments
Start
Evaluations
Customer Evaluation
Engineered System
Engineering
Toward a Completed System
1 - 16

Iterative Refinement

First Loop

- Start at the center of the spiral; plan the project and gather initial requirements
- Perform a risk analysis based on these initial requirements; make a go/no go decision; continue if go
- Create an initial prototype of the system
- Customer (and developer) evaluate the prototype

Second Loop

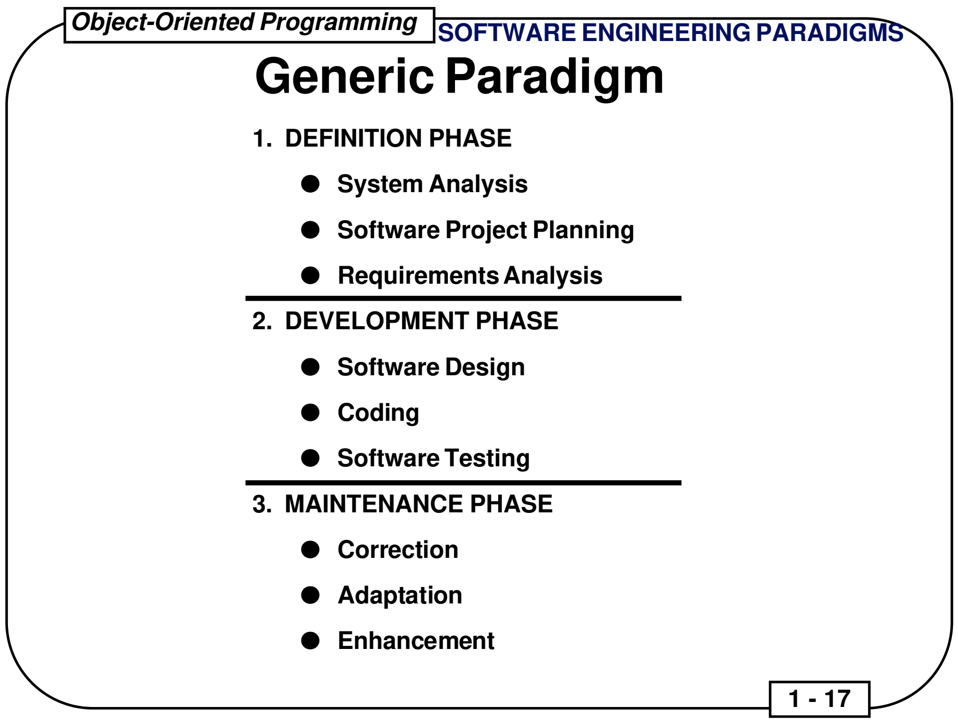
- Feedback from the evaluation is used to refine the requirements and more project planning is done
- Perform a second risk analysis based on the revised requirements; make a go/no go decision; continue if go
- Create a second prototype, based either on the initial prototype or built from scratch
- Customer (and developer) evaluate the second prototype

Nth Loop

- Repeat the Second Loop as desired

After Last Go/No Go Decision

- Engineer the system


Object-Oriented Programming SOFTWARE ENGINEERING PARADIGMS

Generic Paradigm

1. DEFINITION PHASE
   - System Analysis
   - Software Project Planning
   - Requirements Analysis
2. DEVELOPMENT PHASE
   - Software Design
   - Coding
   - Software Testing
3. MAINTENANCE PHASE
   - Correction
   - Adaptation
   - Enhancement

Common Phases for All Methods

Definition Phase

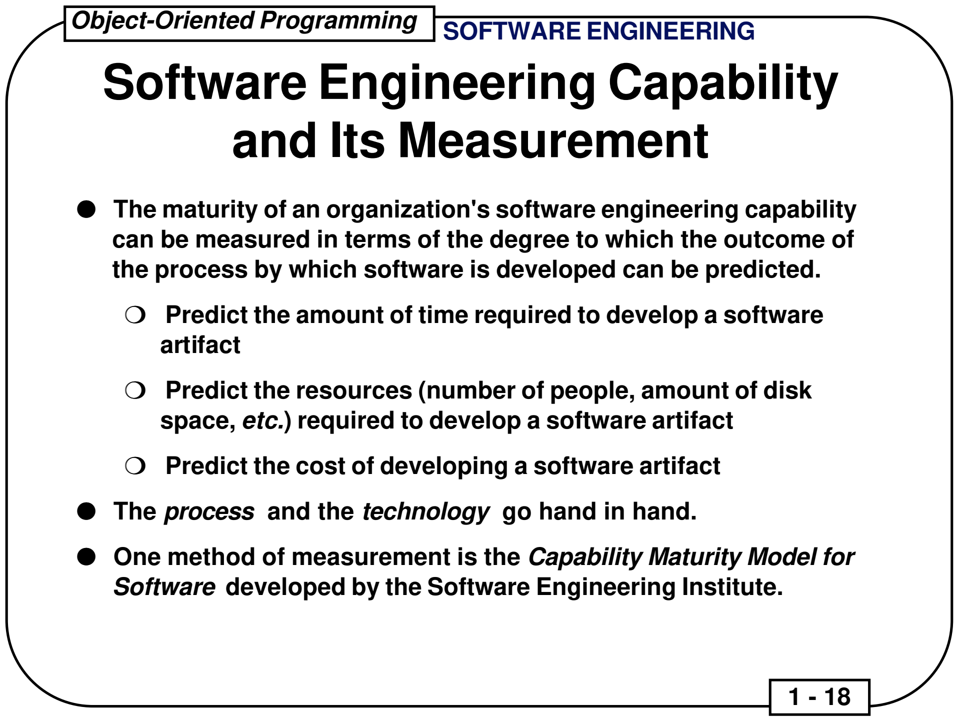
- All methods involve an analysis of the system in which the software resides, the gathering of the requirements for the software, and the planning of the development of the software
- Plan the development and get an initial understanding of the requirements

Development Phase

- Design, code, and test the software

Maintenance Phase

- Support the software after it is released to the customer; there are often three kinds of maintenance to be performed:
  - **Corrective Maintenance** - fix defects uncovered in the software
  - **Adaptive Maintenance** - change the software to run under different environments, such as new versions of an operating system
  - **Enhancement** - extend the capabilities of the software



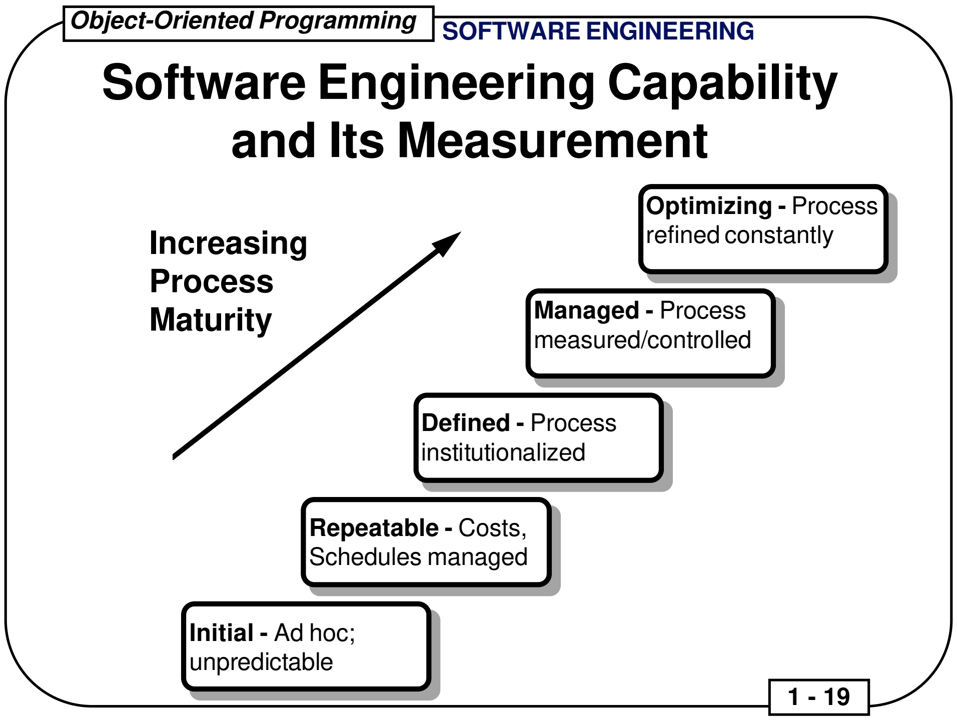
Software Engineering Capability and Its Measurement

- **The maturity of an organization's software engineering capability can be measured in terms of the degree to which the outcome of the process by which software is developed can be predicted.**
  - ❍ **Predict the amount of time required to develop a software artifact**
  - ❍ **Predict the resources (number of people, amount of disk space, *etc.*) required to develop a software artifact**
  - ❍ **Predict the cost of developing a software artifact**
- **The *process* and the *technology* go hand in hand.**
- **One method of measurement is the *Capability Maturity Model for Software* developed by the Software Engineering Institute.**

Capability Maturity Model for Software

This model is defined in two papers from the Software Engineering Institute:

- Paulk, Curtis, Chrissis, *et al*, **Capability Maturity Model for Software**, August, 1991, Report Number CMU/SEI-91-TR-24 and ESD-TR-91-24, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213
- Weber, Paulk, Wise, Withey, *et al*, **Key Practices of the Capability Maturity Model**, August, 1991, Report Number CMU/SEI-91-TR-25 and ESD-TR-91-25, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213



Software Engineering Capability and Its Measurement

Increasing Process Maturity
Optimizing - Process refined constantly
Managed - Process measured/controlled
Defined - Process institutionalized
Repeatable - Costs, Schedules managed
Initial - Ad hoc; unpredictable

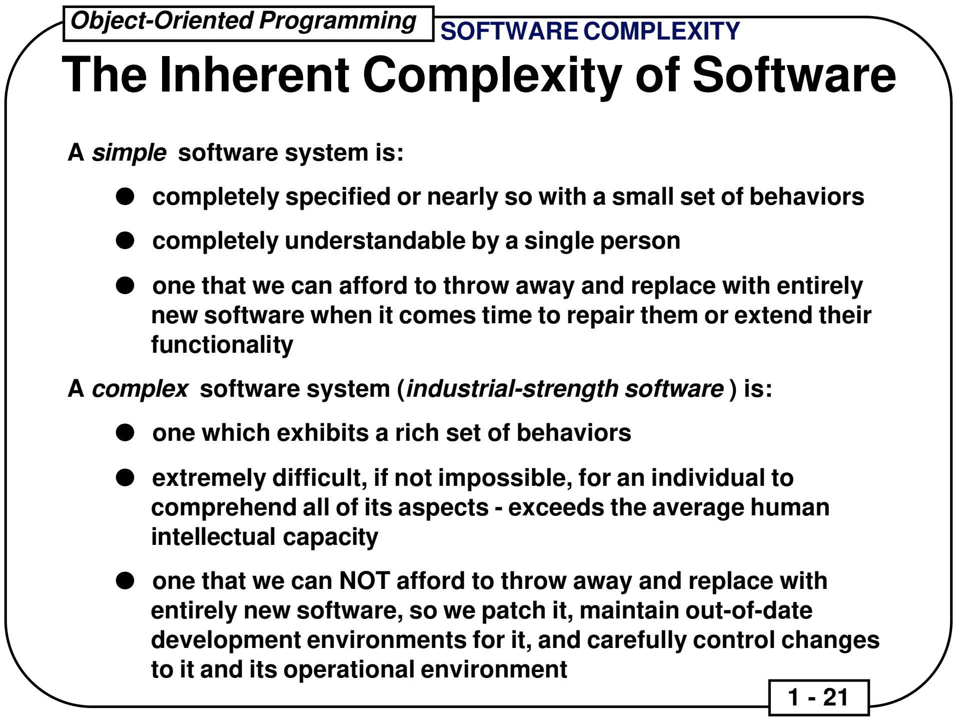
Some Aspects of Each Level

- Level 1: **Initial**
  - Project outcomes are characterized by frequent cost and schedule overruns
  - People are burnt out in the attempt to meet the schedule
- Level 2: **Repeatable**
  - Controls, software quality assurance, and baseline management are in place
  - No commitments are made without thorough review
  - Given experience with one type of project, probability of repeating the level of performance (cost, schedule, and quality) on another similar project is high
- Level 3: **Defined**
  - Process for each project is defined in writing at the outset
  - SQA monitors compliance with standards and is empowered to intervene
  - Project outcomes become more predictable across a broader range of projects
- Level 4: **Managed**
  - Quantitative quality and productivity goals are set for each step in the process
  - High predictability is achieved for each step of the process
- Level 5: **Optimizing**
  - Data collected are used to identify weakness and bottlenecks in the process
  - Causes of errors are analyzed, and future errors prevented

SOFTWARE COMPLEXITY, OBJECT-ORIENTED REQUIREMENTS ANALYSIS (OORA), AND OBJECT-ORIENTED DESIGN (OOD)

- ✓ **The Inherent Complexity of Software**
- ✓ **The Attributes of Complex Systems**
- ✓ **Canonical Form of a Complex System**
- ✓ **On Designing Complex Systems**

- *The Nature of Software*
- *History of Software Development*
- *Software Engineering Paradigms and Technology*
- **Software Complexity, Object-Oriented Requirements Analysis (OORA), and Object-Oriented Design (OOD)**



The Inherent Complexity of Software

A *simple* software system is:

- **completely specified or nearly so with a small set of behaviors**
- **completely understandable by a single person**
- **one that we can afford to throw away and replace with entirely new software when it comes time to repair them or extend their functionality**

A *complex* software system (*industrial-strength software*) is:

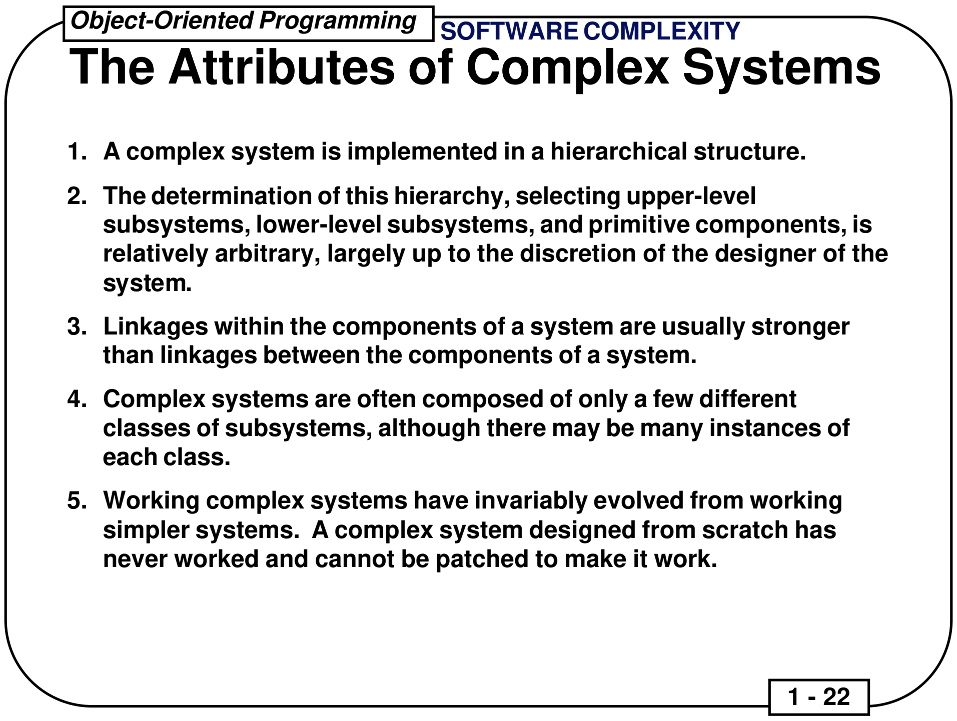
- **one which exhibits a rich set of behaviors**
- **extremely difficult, if not impossible, for an individual to comprehend all of its aspects - exceeds the average human intellectual capacity**
- **one that we can NOT afford to throw away and replace with entirely new software, so we patch it, maintain out-of-date development environments for it, and carefully control changes to it and its operational environment**

Some software systems are small, useful, and easily addressed by an individual. They are not of concern to us in this class since they are easily developed using ad-hoc or conventional, non-object-oriented software engineering approaches. Object-oriented technology developed out of a need to be able to handle complex software systems.

People of the genius class will always be around, demonstrating extraordinary skills in developing larger software systems. These are the people we want to employ as the system architects of our complex software systems. However, the world is only sparsely populated with geniuses. There is a touch of genius in all of us, but it cannot be relied upon in the development of industrial-strength software. Object-oriented technology is a more disciplined approach to mastering the complexity of industrial-strength software without having to rely on the divine inspiration of genius.

Why is industrial-strength software so complex?

- The problem domain itself is complex. Consider the air traffic control system of the United States or the telephone system of AT&T.
- Managing the development process itself is a complex problem. A few decades ago, our software consisted largely of assembly language programs that were only a few thousand lines of code long. Today, delivered software systems range in size from a few hundred thousand lines of code to millions or tens of millions of lines of code developed by teams of 50, 100, 1000, or more people.
- Software affords almost too much flexibility. This flexibility is seductive, but it has a drawback in that there is a lack of standards which support extensive reuse without resorting to hand-crafting the software.
- Software systems are discrete rather than continuous. The larger the system, the more of an explosion of states we have (often exponential).



The Attributes of Complex Systems

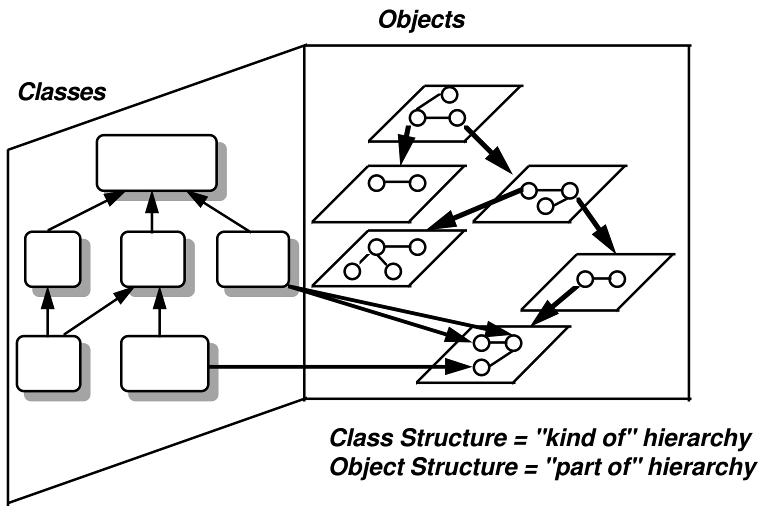
1. **A complex system is implemented in a hierarchical structure.**
2. **The determination of this hierarchy, selecting upper-level subsystems, lower-level subsystems, and primitive components, is relatively arbitrary, largely up to the discretion of the designer of the system.**
3. **Linkages within the components of a system are usually stronger than linkages between the components of a system.**
4. **Complex systems are often composed of only a few different classes of subsystems, although there may be many instances of each class.**
5. **Working complex systems have invariably evolved from working simpler systems. A complex system designed from scratch has never worked and cannot be patched to make it work.**

Two kinds of complex systems are:

decomposable - one which may be divided into identifiable, independent parts

nearly decomposable - one which may be divided into identifiable parts, but the parts are not completely independent

Canonical Form of a Complex System


Objects
Classes
Class Structure = "kind of" hierarchy
Object Structure = "part of" hierarchy

Most complex systems do not embody a single hierarchy. Complex systems are usually composed of a network of related hierarchies. Two broad types of hierarchies exist:

- A "part of" hierarchy, also known as the *object structure*. For example, an aircraft is composed of a propulsion system, a flight-control system, and so on. These systems are the major parts of the aircraft.
- A "kind of" hierarchy, also known as the *class structure*. For example, a turbofan engine is a kind of jet engine for the aircraft, and a high-bypass turbofan engine is a kind of turbofan engine. A particular high-bypass turbofan engine, ID number 20943G56, is an instance of the class of all high-bypass turbofan engines.

A successful complex software system encompasses:

- a well-engineered class structure
- a well-engineered object structure
- the five attributes of a complex system

On Designing Complex Systems

Requirements Analysis **- the disciplined approach used to understand a problem**

Design **- the disciplined approach used to devise a solution to a problem**

The Purpose of Design

To construct a system that:

- **satisfies a given specification**
- **conforms to limitations of the target**
- **meets constraints on performance and resource usage**
- **satisfies a given set of design criteria on the artifact**
- **satisfies restrictions on the design process itself, such as cost and schedule**

Elements of Design

Notation **- the language of expression**

Process **- the steps taken for the orderly construction of the design**

Tools **- the artifacts that support the design process by reducing the level of effort**