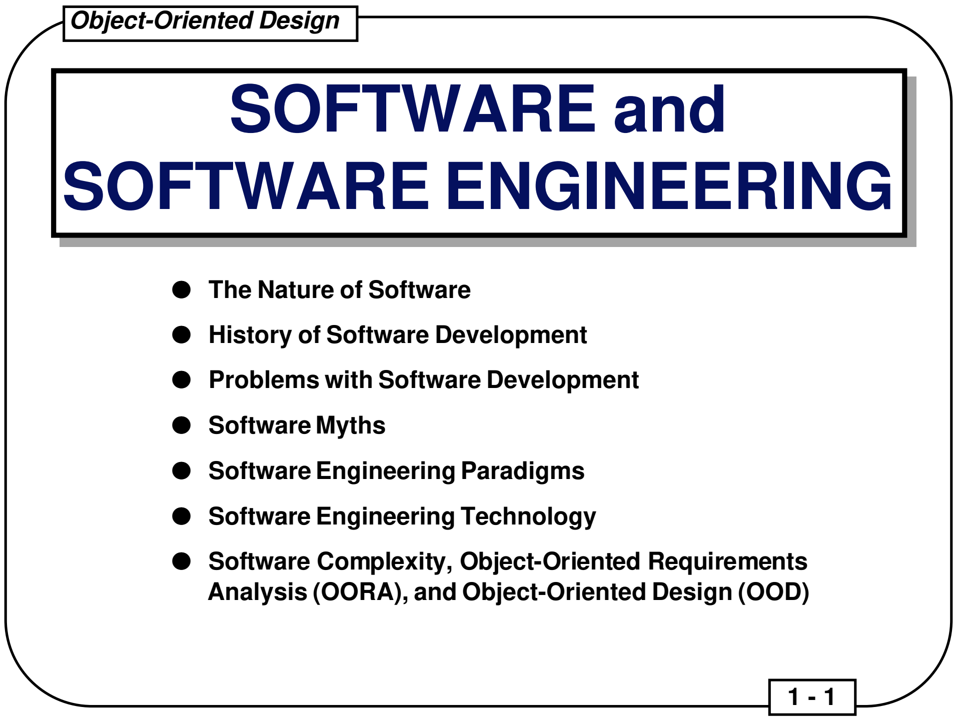

Object-Oriented Design

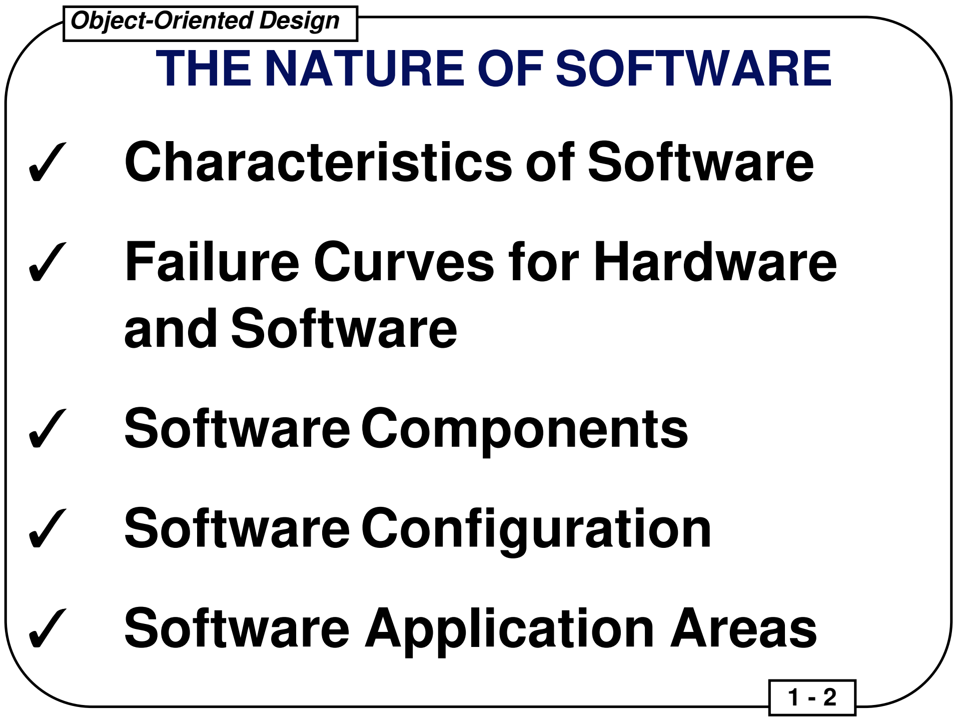
SOFTWARE and SOFTWARE ENGINEERING

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

1 - 1

Objectives of Module 1

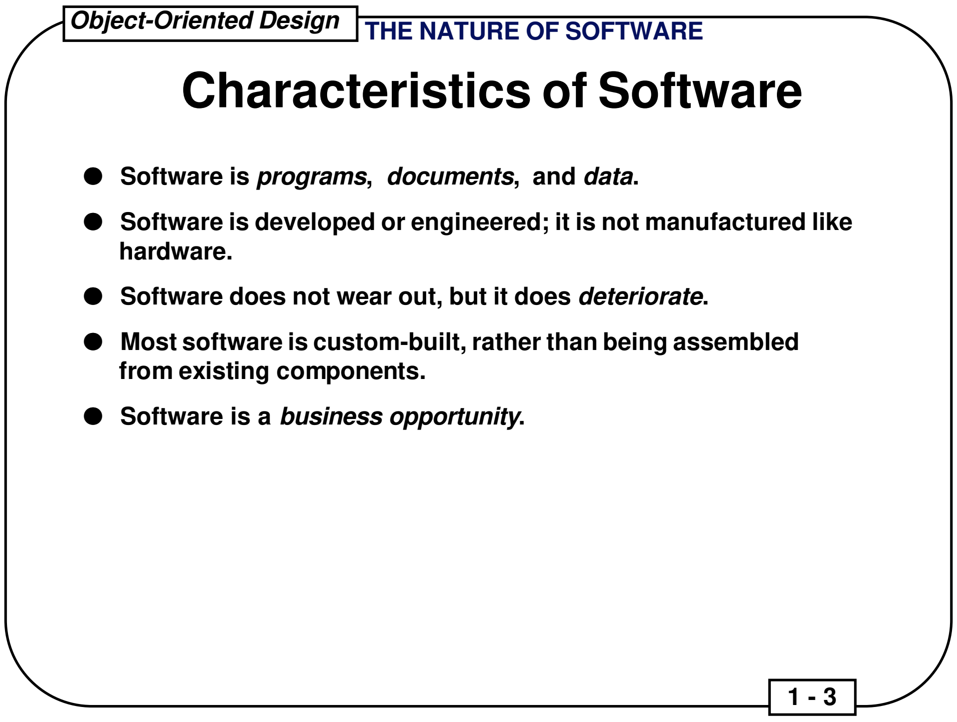
- Present and discuss the idea that software is much more than just code -- *engineered* software is composed of *controlled configuration items* which include documents, data, and code
- Present and discuss the history of software development, including its evolution into a business
- Present and discuss many of the problems with doing software development, particularly when there is more than one person involved
- Present and discuss many myths about software development and explain why some of these myths are fallacies
- Present and discuss several different software engineering paradigms, showing different methods for developing engineered software:
  - Classic "waterfall" method
  - Rapid prototyping
  - Spiral method
  - Fourth generation method
- Present and discuss some of the technologies used in the support of software engineering:
  - *Computer-Aided Software Engineering* (CASE)
  - Ada programming language
- Introduce the concepts of complexity, OORA, and OOD



THE NATURE OF SOFTWARE

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

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

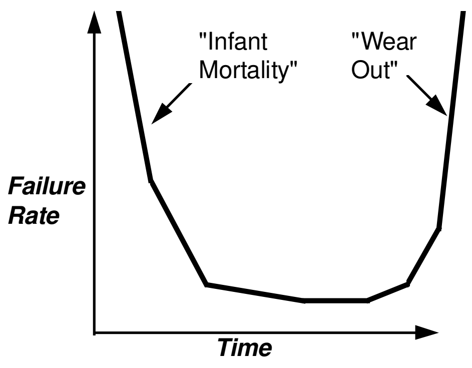


Characteristics of Software

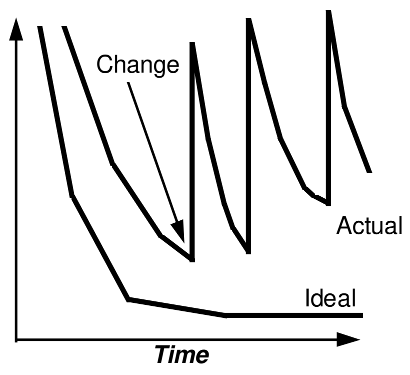
- Software is *programs*, *documents*, and *data*.
- Software is developed or engineered; it is not manufactured like hardware.
- Software does not wear out, but it does *deteriorate*.
- Most software is custom-built, rather than being assembled from existing components.
- Software is a *business opportunity*.

1. 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



FAILURE CURVE FOR HARDWARE



FAILURE CURVE FOR 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.

Moral

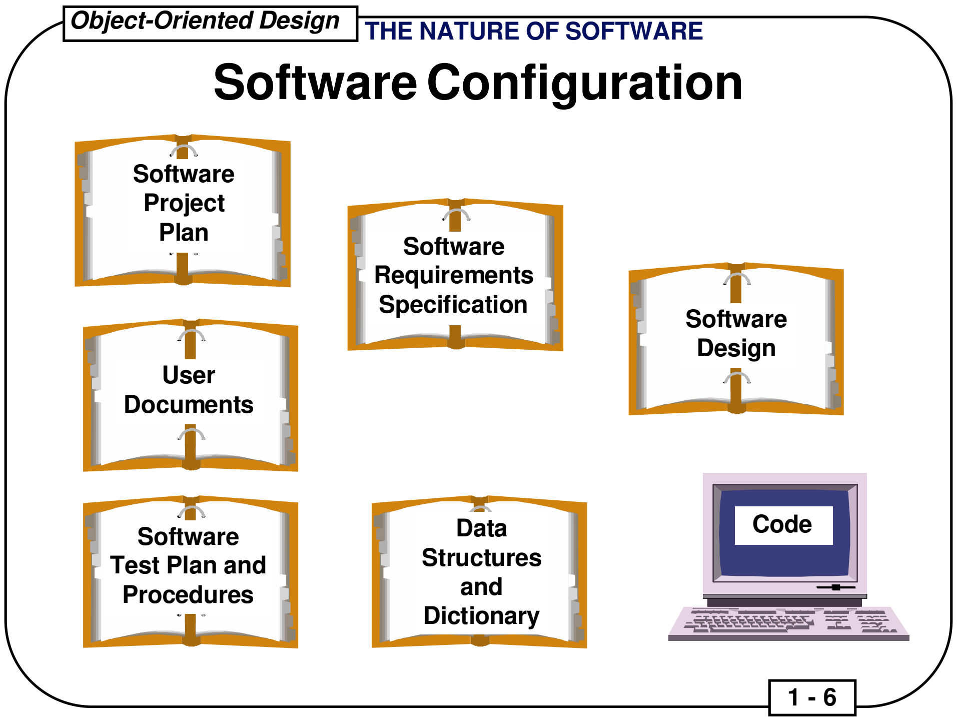
Don't buy extended warranty contracts.

Standard warranty is usually long enough to pass through the infant mortality period.

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 Design

THE NATURE OF SOFTWARE

Software Configuration

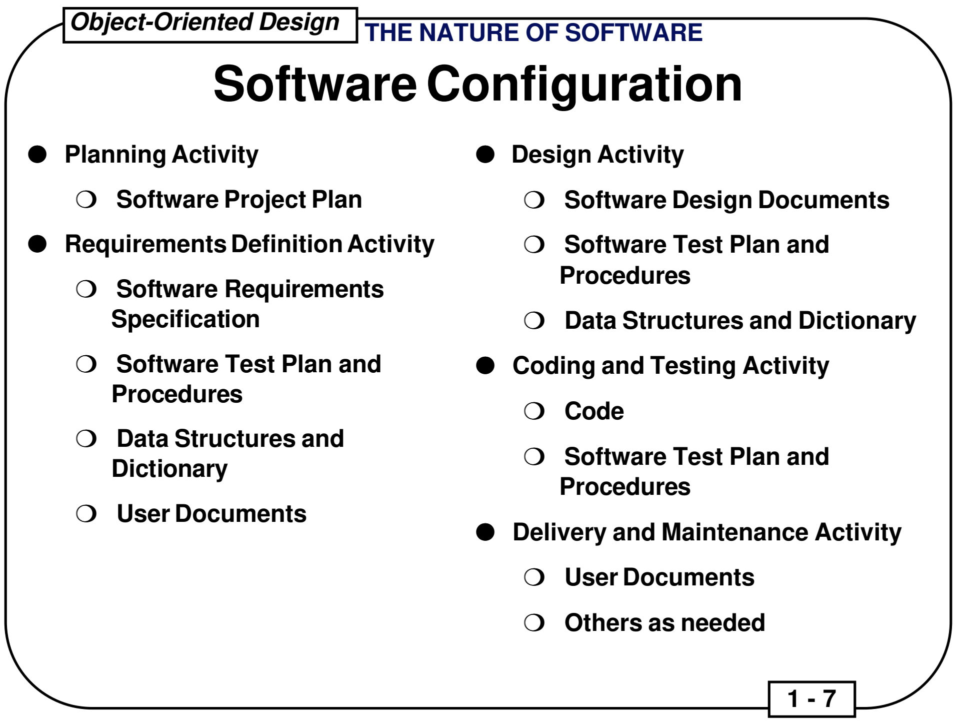
Software Project Plan
Software Requirements Specification
Software Design
User Documents
Software Test Plan and Procedures
Data Structures and Dictionary
Code

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.

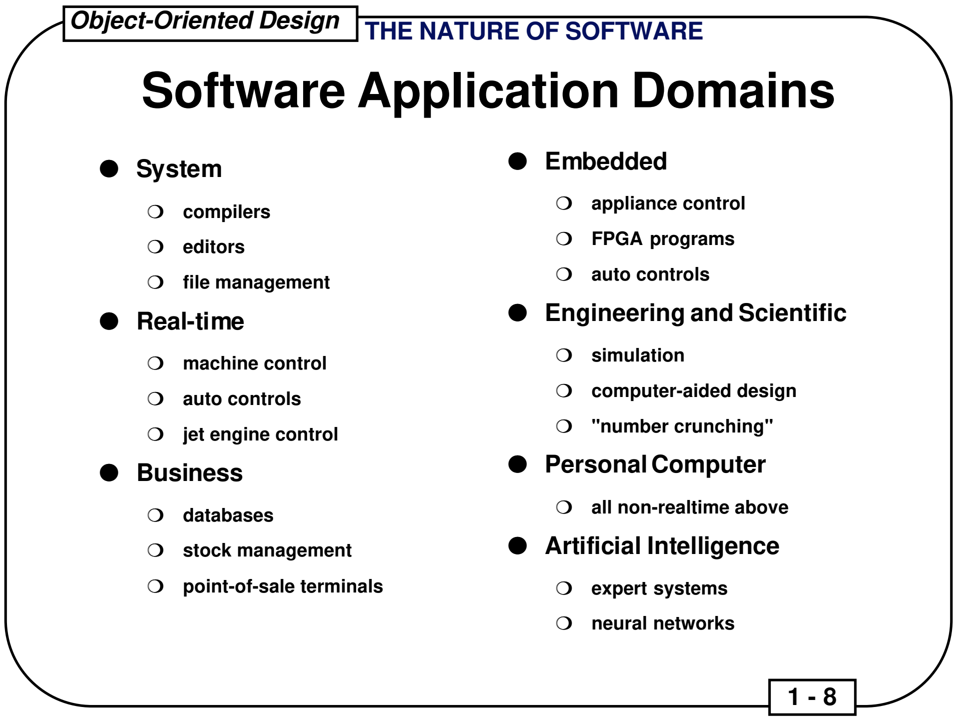


Software Configuration

- **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.



Software Application Domains

- **System**
  - ❍ compilers
  - ❍ editors
  - ❍ file management
- **Real-time**
  - ❍ machine control
  - ❍ auto controls
  - ❍ jet engine control
- **Business**
  - ❍ databases
  - ❍ stock management
  - ❍ point-of-sale terminals
- **Embedded**
  - ❍ appliance control
  - ❍ FPGA programs
  - ❍ auto controls
- **Engineering and Scientific**
  - ❍ simulation
  - ❍ computer-aided design
  - ❍ "number crunching"
- **Personal Computer**
  - ❍ all non-realtime above
- **Artificial Intelligence**
  - ❍ expert systems
  - ❍ neural networks

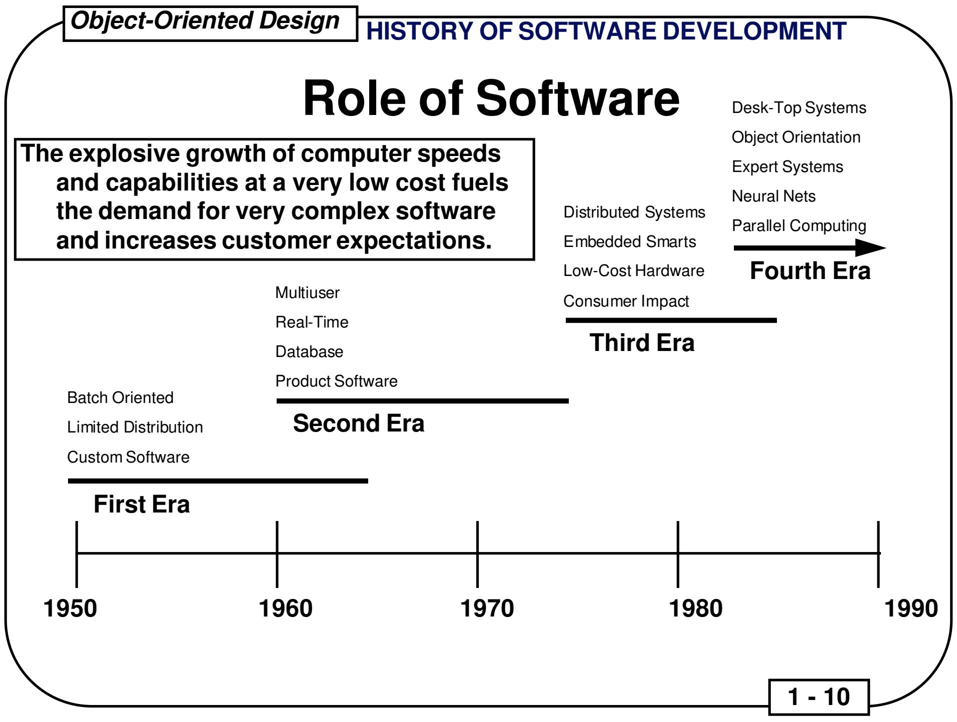
There are many, many diverse application domains in which software is being developed, and, for each domain and each organization within each domain, there are many, many different ways to develop this software:

- ad hoc, which is by far the most common
- using different accepted engineering methodologies, such as
  - ❍ the classic "waterfall" approach
  - ❍ rapid prototyping
  - ❍ fourth generation techniques
  - ❍ the spiral model
  - ❍ a combination of the above
- using different sets of procedures, which include
  - ❍ documentation standards
  - ❍ coding standards
  - ❍ test standards
  - ❍ procedures for estimating cost and schedule

HISTORY OF SOFTWARE DEVELOPMENT

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

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



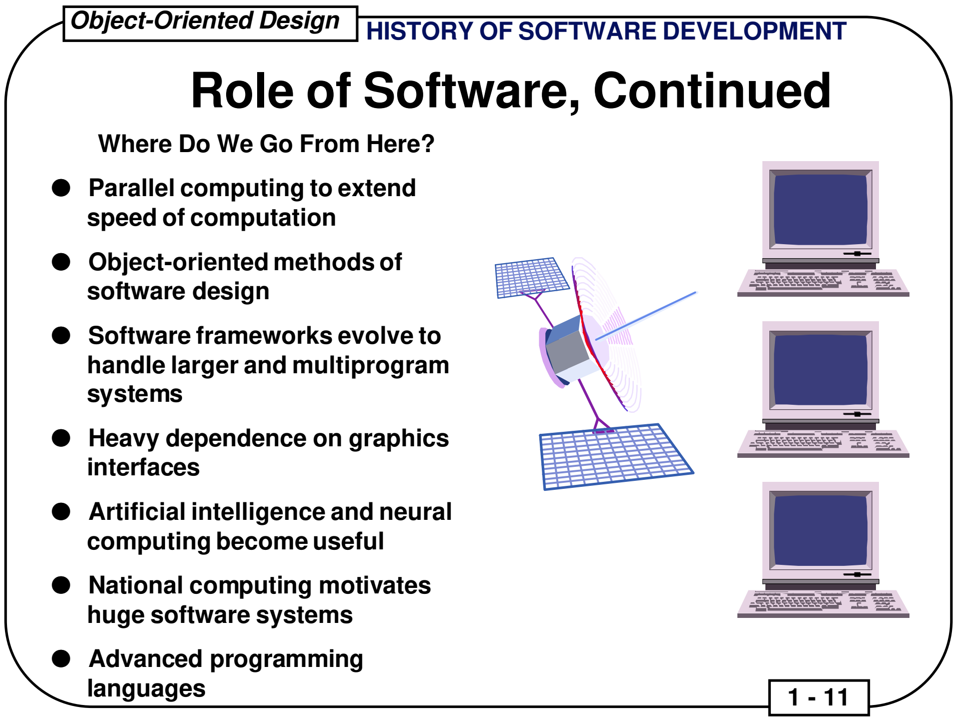
Object-Oriented Design HISTORY OF SOFTWARE DEVELOPMENT

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 - 10

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



Role of Software, Continued

Where Do We Go From Here?

- **Parallel computing to extend speed of computation**
- **Object-oriented methods of software design**
- **Software frameworks evolve to handle larger and multiprogram systems**
- **Heavy dependence on graphics interfaces**
- **Artificial intelligence and neural computing become useful**
- **National computing motivates huge software systems**
- **Advanced programming languages**

- One concept which dominates all of these ideas is that high quality software is required in all cases.
- The software engineering community has learned that two things are needed to develop high quality software:
  - ❍ a good software development process
  - ❍ technological innovations which support the selected process
- Technology alone, without the process, is not enough and often adds to the risk and the problems rather than reducing the risk and the problems.

Industrial View

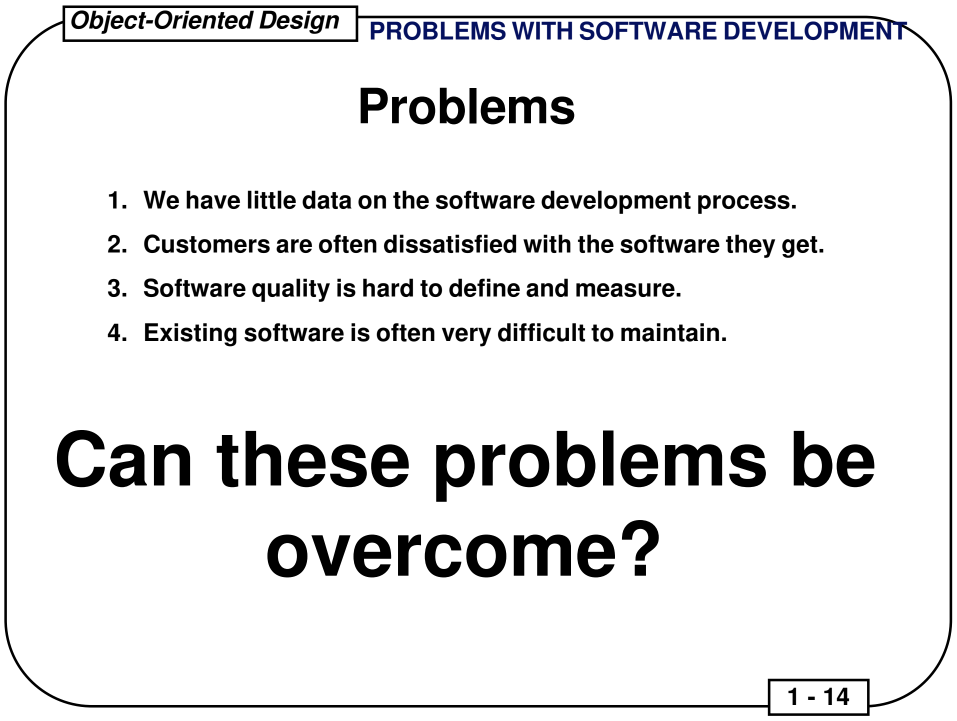
- **Why does it take so long to finish a working software system?**
- **Why are development costs so high?**
- **Why can't we find all software errors before software is delivered?**
- **How can we measure the progress of software development?**
- **How can we survive in the global economy?**

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

PROBLEMS WITH SOFTWARE DEVELOPMENT

- ✓ **Problems**
- ✓ **Causes**

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



Problems

1. **We have little data on the software development process.**
2. **Customers are often dissatisfied with the software they get.**
3. **Software quality is hard to define and measure.**
4. **Existing software is often very difficult to maintain.**

Can these problems be overcome?

- Historically, measurements of software development were not done, so we did not gather any data from past experiences to use in predicting the schedules and costs of future projects.
- Customer needs were usually understood only vaguely. Consequently, programs often fell short of the customers' desires.
- A solid quantification of the metrics associated with the software does not exist, so it becomes difficult to predict software quality.
- Maintenance has become the most expensive, difficult, and poorly planned task of the entire software life cycle.

Causes

- **No spare parts to replace, so an error in the original software is also in every copy.**
- **Software quality is a human problem.**
- **Project managers often have no software development experience.**
- **Software developers often have little or no formal training in engineering the development of the software product.**
- **Resistance to change from programming as an art to programming as an engineering task can be significant.**

SOFTWARE MYTHS

✓ **Customer Myths**

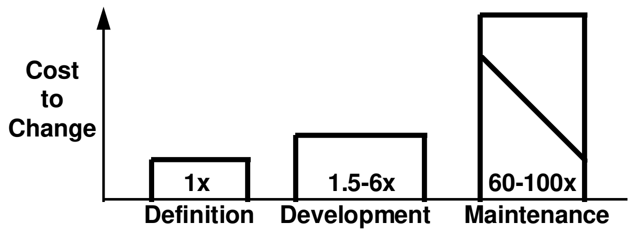
✓ **Developer Myths**

✓ **Management Myths**

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

Customer Myths

Myth	*Reality*
● A general statement of objectives is enough to get going. Fill in the details later.	● Poor up-front definition of the requirements is *THE* major cause of poor and late software.
● Project requirements continually change, but change can be easily accommodated because software is flexible.	● Cost of the change to software in order to fix an error increases dramatically in later phases of the life of the software.

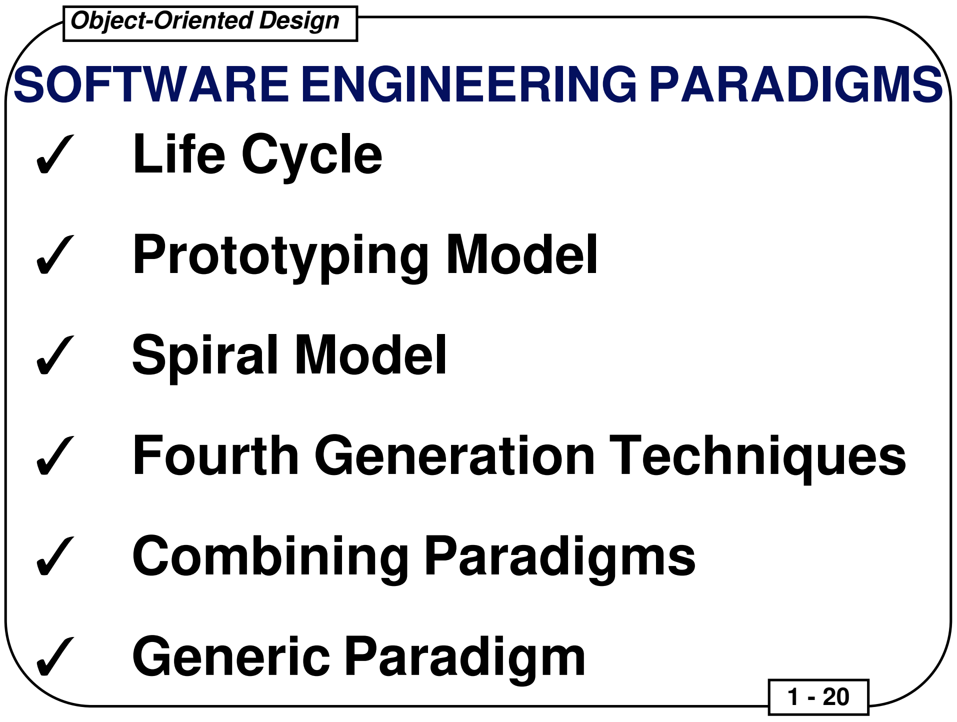


Developer Myths

Myth	*Reality*
● **Once a program is written and works, the developer's job is done.**	● **50%-70% of the effort expended on a program occurs after it is delivered to the customer.**
● **Until a program is running, there is no way to assess its quality.**	● **Software reviews can be more effective in finding errors than testing for certain classes of errors.**
● **The only deliverable for a successful project is a working program.**	● **A software configuration includes documentation, regeneration files, test input data, and test results data.**

Management Myths

Myth	*Reality*
Books of standards exist in -house so software will be developed satisfactorily.	Books may exist, but they are usually not up to date and not used.
Computers and software tools that are available in-house are sufficient.	CASE tools are needed but are not usually obtained or used.
We can always add more programmers if the project gets behind.	"Adding people to a late software project makes it later." -- *Brooks*



Object-Oriented Design

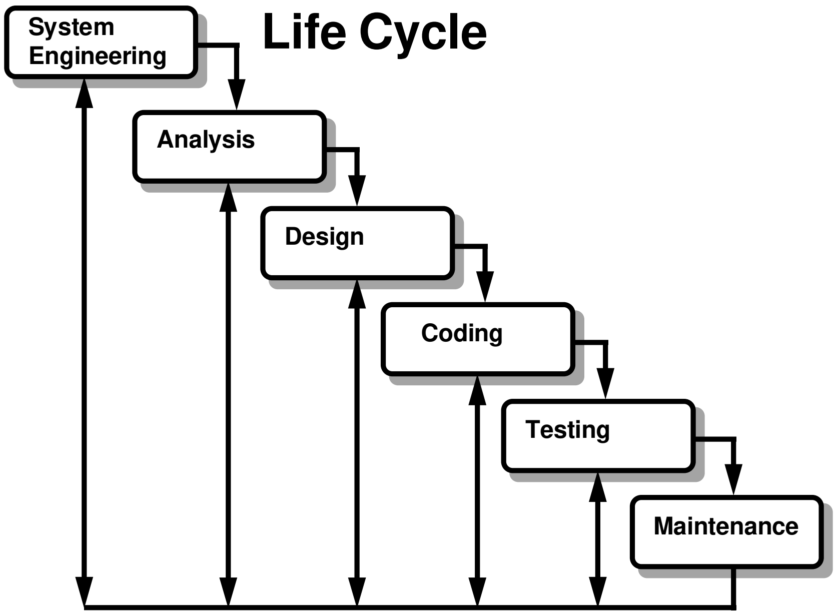
SOFTWARE ENGINEERING PARADIGMS

- ✓ **Life Cycle**
- ✓ **Prototyping Model**
- ✓ **Spiral Model**
- ✓ **Fourth Generation Techniques**
- ✓ **Combining Paradigms**
- ✓ **Generic Paradigm**

1 - 20

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

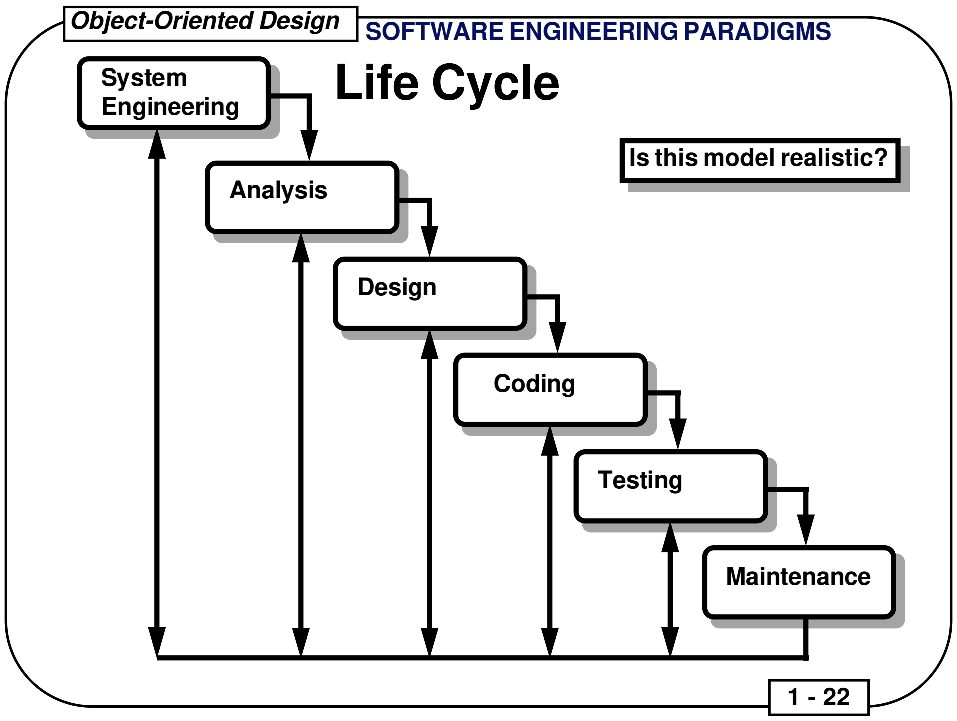
Life Cycle



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



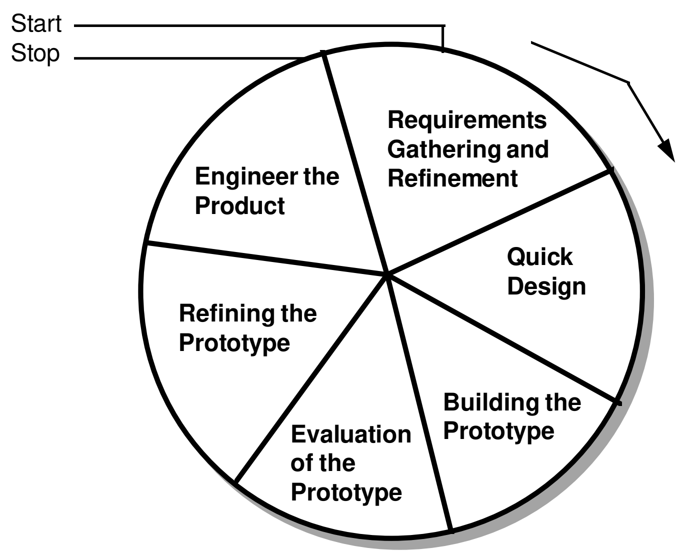
Life Cycle

Is this model realistic?

Problems with the Classic "Waterfall" Model

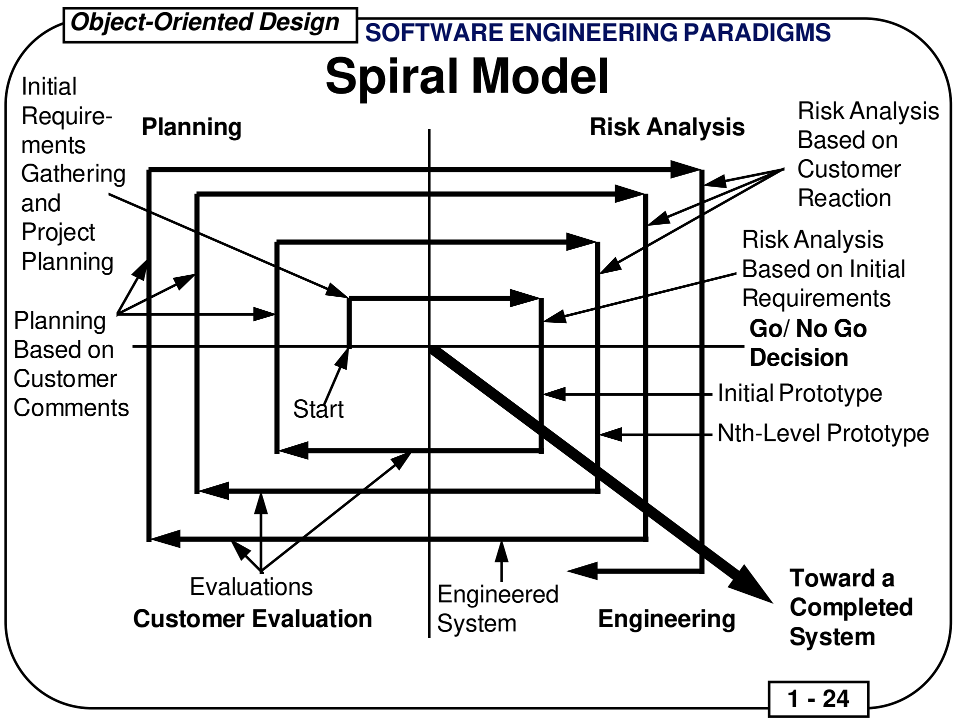
- Real projects rarely follow strict sequential development.
- It is very difficult to fully state all the requirements up front. The customer does not often know exactly what his requirements are or does not provide all the necessary input to fully state the requirements.
- This model demands patience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



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
Planning Based on Customer Comments
Start
Initial Prototype
Nth-Level Prototype
Evaluations
Customer Evaluation
Engineered System
Engineering
Toward a Completed System

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

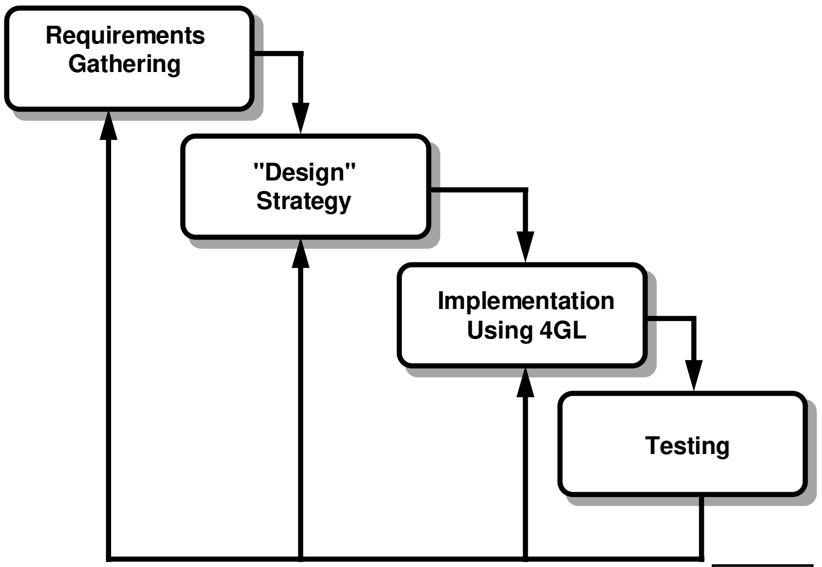
Nth Loop

- Repeat the Second Loop as desired

After Last Go/No Go Decision

- Engineer the system

Fourth Generation Techniques



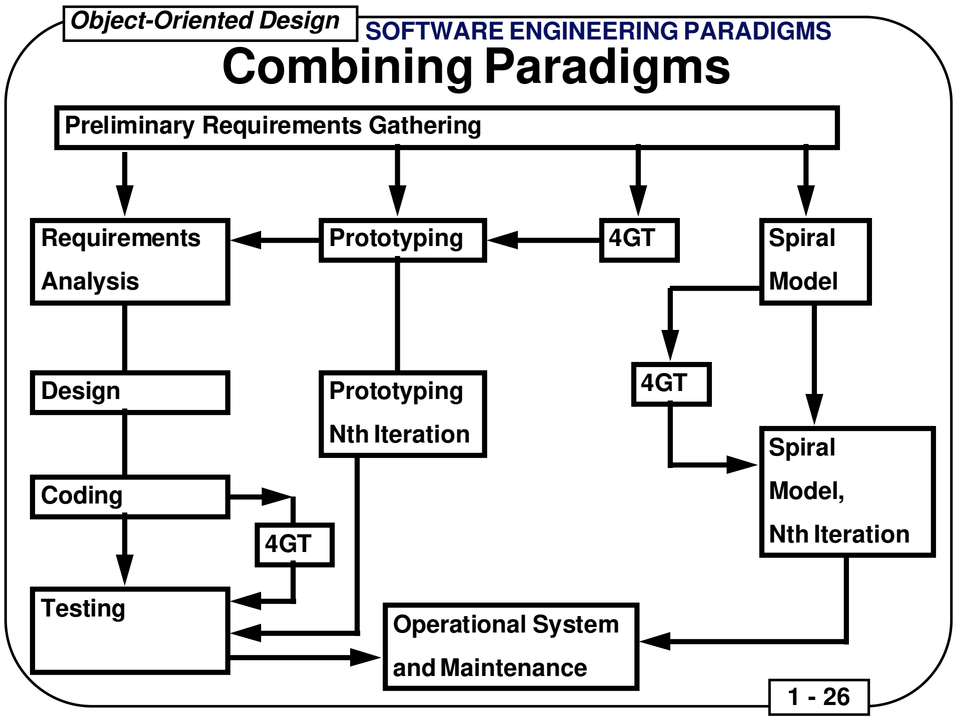
An Enhancement to the Classic "Waterfall"

First Pass

- Requirements Gathering - Collection of requirements as before
- "Design Strategy" - Design with 4th Generation Languages is often done online and is quite similar to coding
- Implementation Using 4GL - Coding
- Testing - Testing as before, with customer and developer evaluation

Iterations

- Reenter the waterfall where required as indicated by the evaluation



Combining Paradigms

Preliminary Requirements Gathering
Requirements Analysis
Prototyping
4GT
Spiral Model
Design
Prototyping Nth Iteration
4GT
Spiral Model, Nth Iteration
Coding
4GT
Testing
Operational System and Maintenance

Applying Different Paradigms

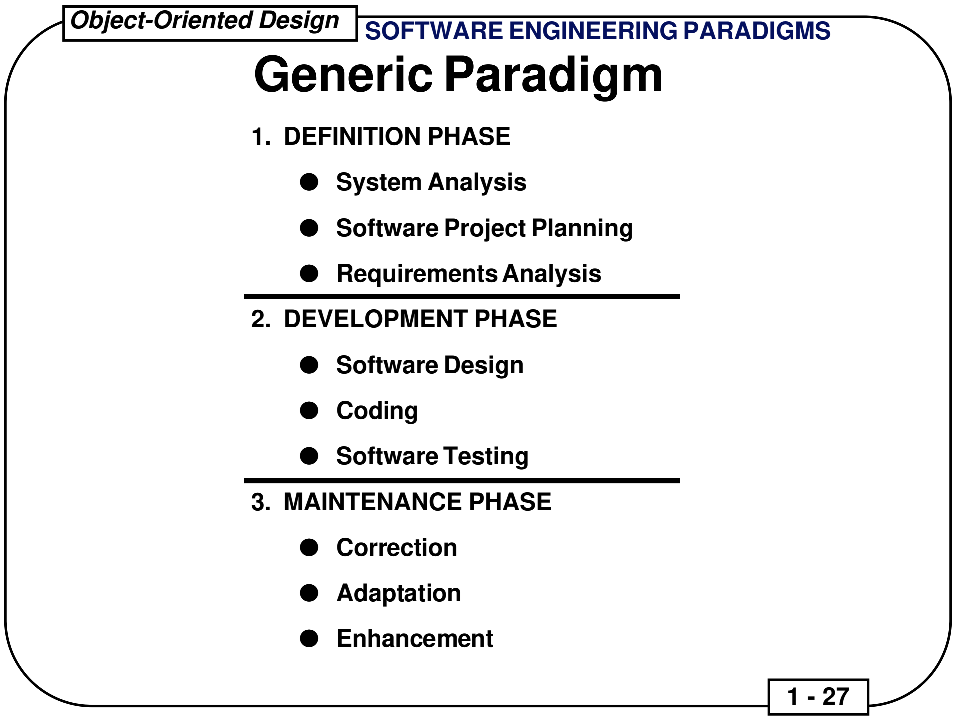
to Different Parts of the System

Preliminary Requirements Gathering

- A preliminary statement of requirements for the entire project is initially obtained
- Based on this statement of requirements, different methods are applied to different parts of the system as is deemed reasonable by the developer and the customer

Movement Through Each Method

- Different parts of the system are developed independently
- Integration may be a high risk area, so integration testing must be thorough



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

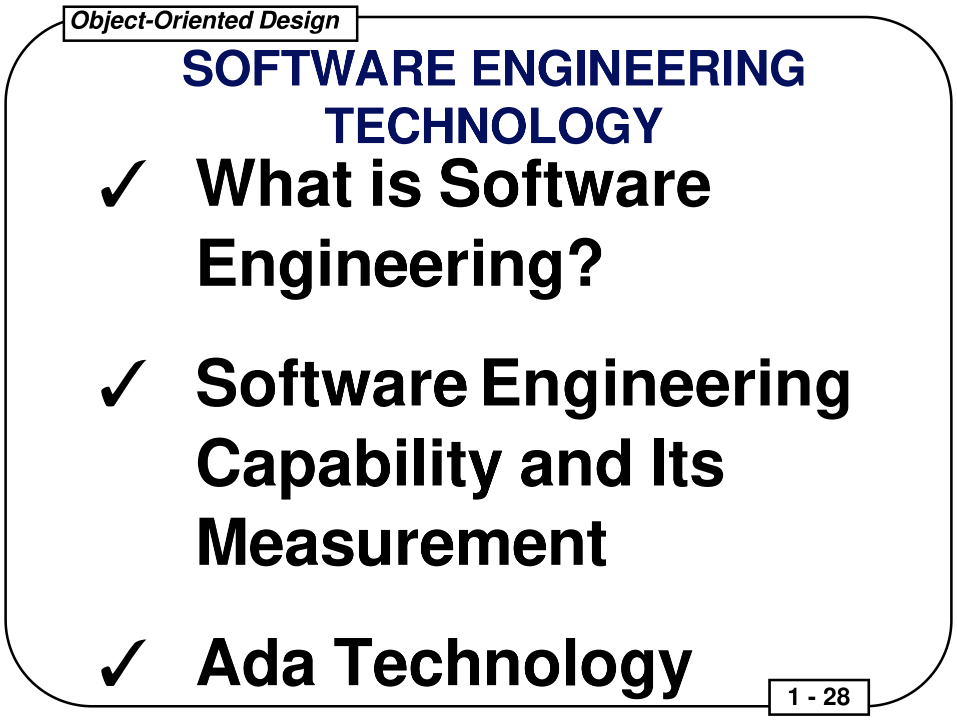
- All methods involve an analysis of the system in which the software resides, the gathering of the requirements for the software, and the planning of the development of the software
- Plan the development and get an initial understanding of the requirements

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



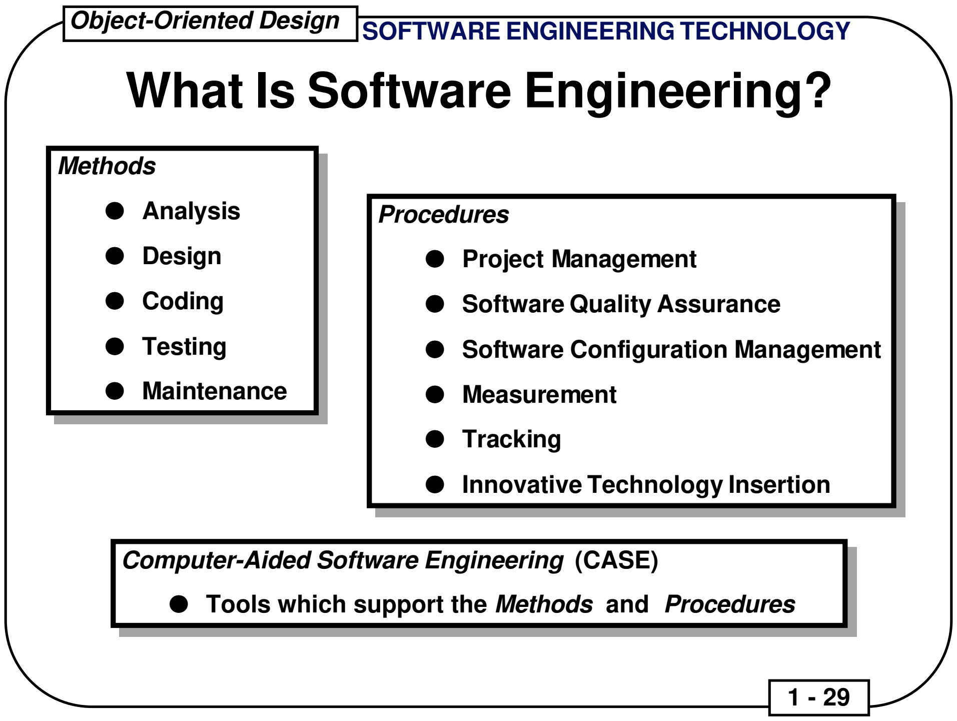
SOFTWARE ENGINEERING TECHNOLOGY

✓ **What is Software Engineering?**

✓ **Software Engineering Capability and Its Measurement**

✓ **Ada Technology**

- *The Nature of Software*
- *History of Software Development*
- *Problems with Software Development*
- *Software Myths*
- *Software Engineering Paradigms*
- **Software Engineering 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

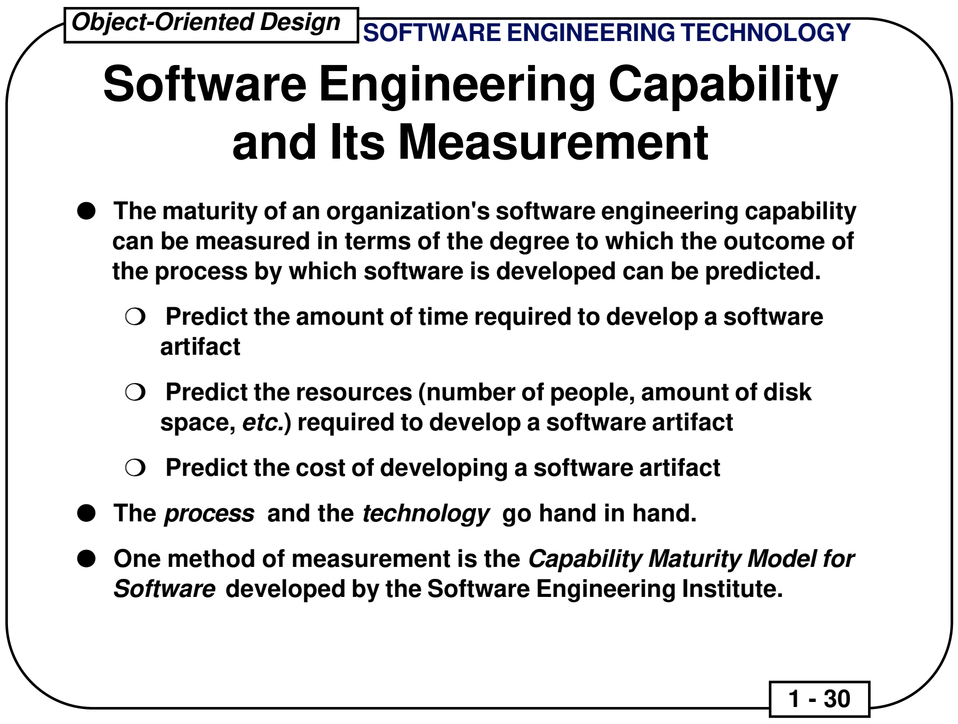
- *Methods* comprise the techniques used to perform the various phases of the software development
- *Methods* are not necessarily documented formally and are often unique to each organization and its culture
- Once a method is selected for a project, automated facilities may come into play to support 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



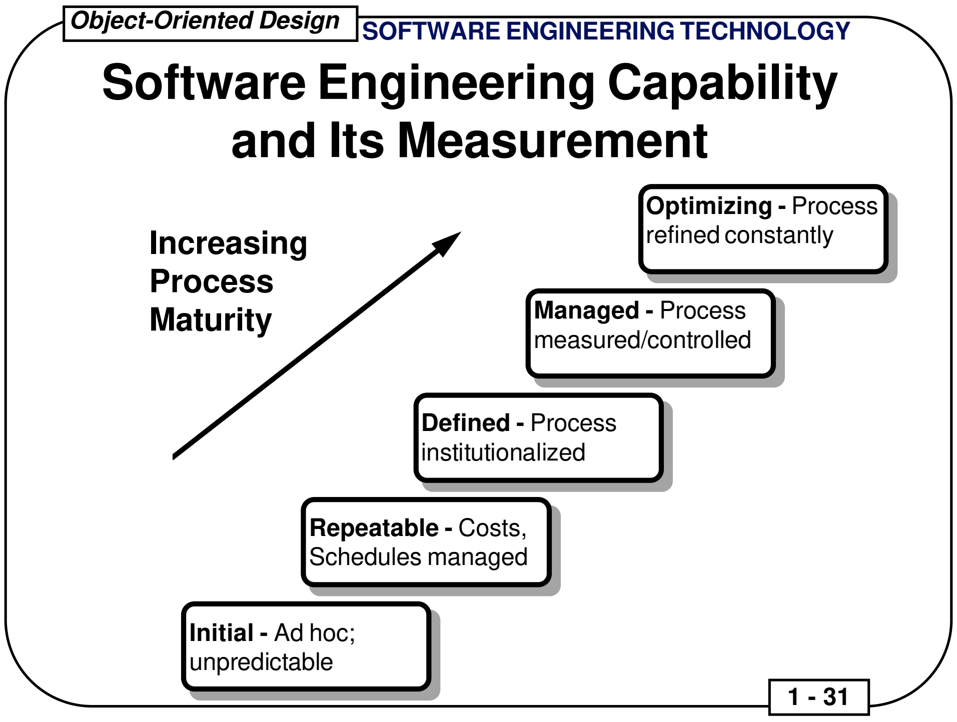
Software Engineering Capability and Its Measurement

- 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



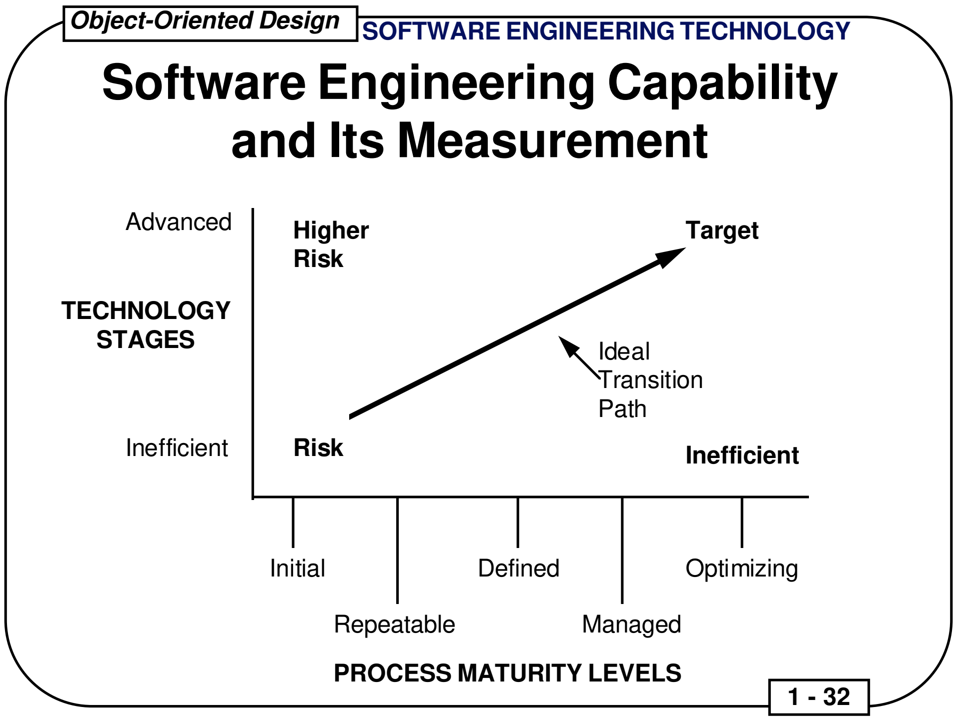
Object-Oriented Design SOFTWARE ENGINEERING TECHNOLOGY

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 achived 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

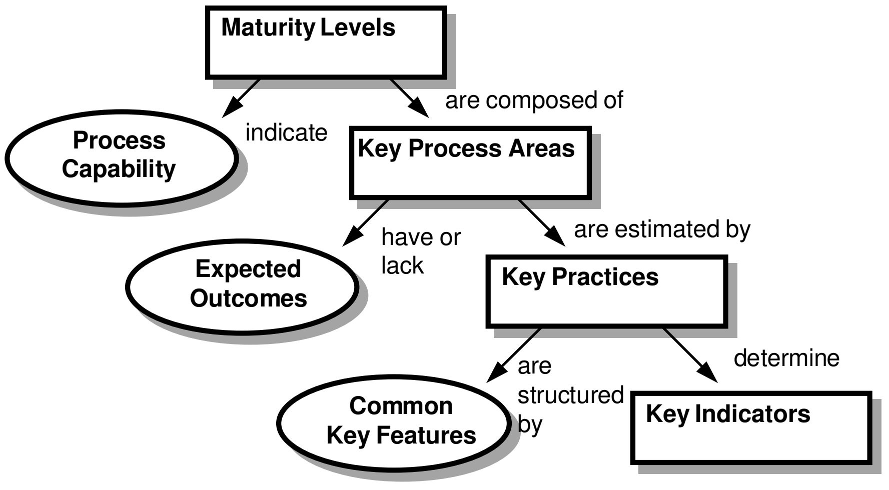


Software Engineering Capability and Its Measurement

State of the Practice

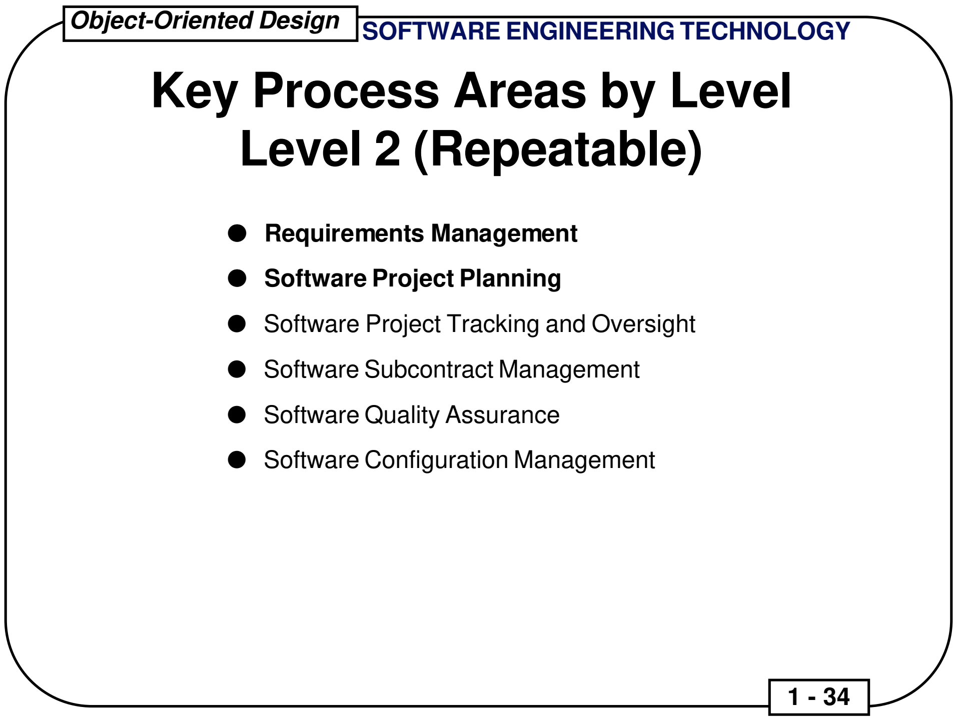
- Corporations in the United States (based on SEI Corporate Affiliates who voluntarily took the CMM evaluation in 1989) --
  - 74% are at Level 1 (Initial)
  - 22% are at Level 2 (Repeatable)
  - 4% are at Level 3 (Defined)
- Corporations in Japan (based on a visit to Japan by an SEI team in 1990) --
  - 95%+ are at Level 1 (Initial)
  - 5%- are at Level 2 (Repeatable)

Software Engineering Capability and Its Measurement



Common Key Features of the Key Practices include --

- Goals
- Commitment to Perform
- Ability to Perform
- Activities Performed
- Monitoring Implementation
- Verifying Implementation

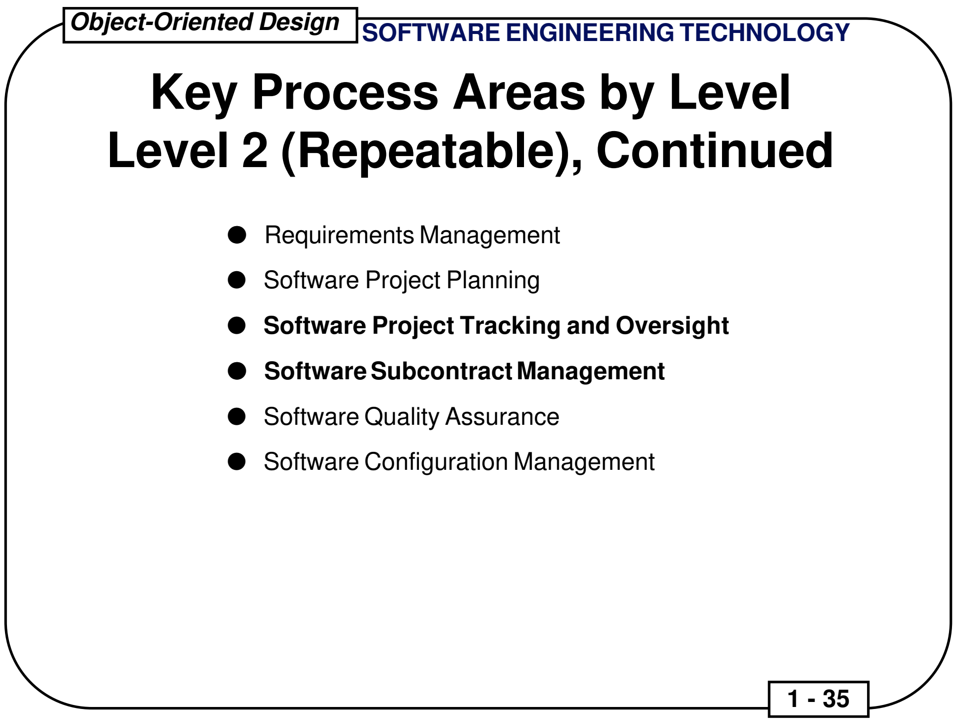


Key Process Areas by Level Level 2 (Repeatable)

- **Requirements Management**
- **Software Project Planning**
- Software Project Tracking and Oversight
- Software Subcontract Management
- Software Quality Assurance
- Software Configuration Management

Goals for Key Process Areas in Level 2

- **Requirements Management**
  - ❍ The system requirements allocated to software provide a clearly stated, verifiable, and testable foundation for software engineering and software management.
  - ❍ The allocated requirements define the scope of the software effort.
  - ❍ The allocated requirements and changes to the allocated requirements are incorporated into the software plans, products, and activities in an orderly manner.
- **Software Project Planning**
  - ❍ A plan is developed that appropriately and realistically covers the software activities and commitments.
  - ❍ All affected groups and individuals understand the software estimates and plans and commit to support them.
  - ❍ The software estimates and plans are documented for use in tracking the software activities and commitments.

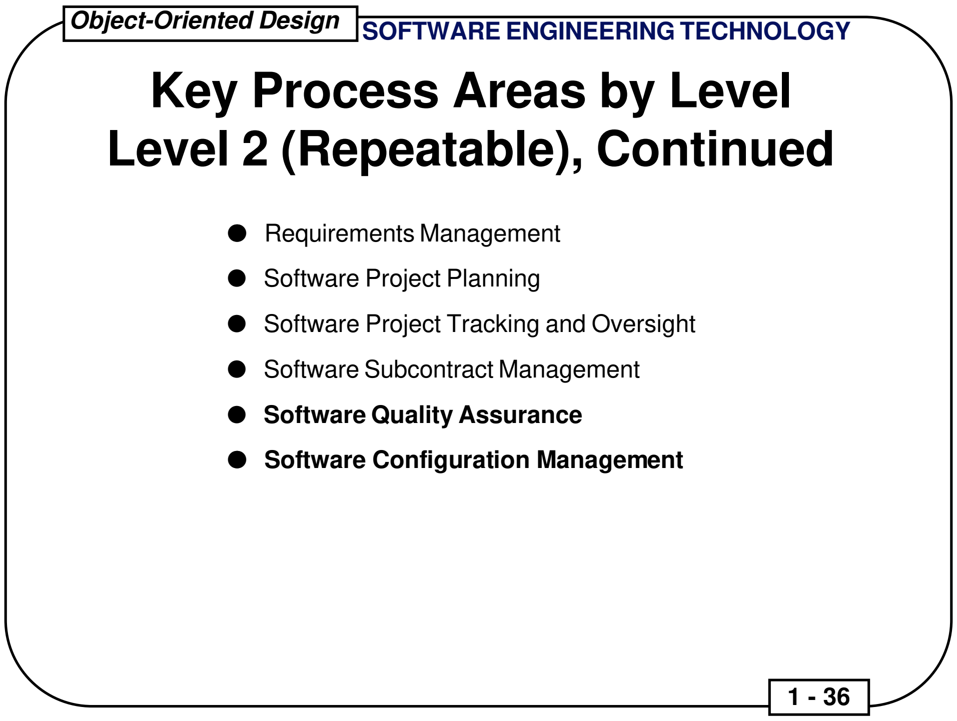


Key Process Areas by Level Level 2 (Repeatable), Continued

- Requirements Management
- Software Project Planning
- **Software Project Tracking and Oversight**
- **Software Subcontract Management**
- Software Quality Assurance
- Software Configuration Management

Goals for Key Process Areas in Level 2

- **Software Project Tracking and Oversight**
  - ❍ Actual results and performance of the software project are tracked against documented and approved plans.
  - ❍ Corrective actions are taken when the actual results and performance of the software project deviate significantly from the plans.
  - ❍ Changes to software commitments are understood and agreed to by all affected groups and individuals.
- **Software Subcontract Management**
  - ❍ The prime contractor selected qualified subcontractors.
  - ❍ The software standards, procedures, and product requirements for the subcontract comply with the prime contractor's commitments.
  - ❍ Commitments between the prime contractor and subcontractor are understood and agreed to by both parties.
  - ❍ The prime contractor tracks the subcontractor's actual results and performance against the commitments.

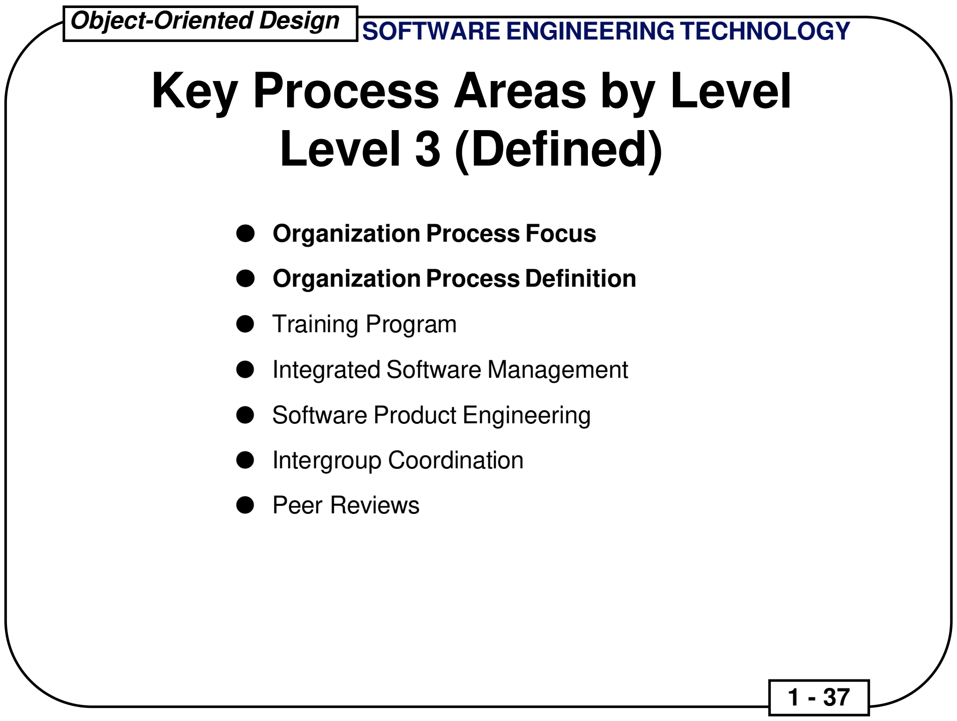


Key Process Areas by Level Level 2 (Repeatable), Continued

- Requirements Management
- Software Project Planning
- Software Project Tracking and Oversight
- Software Subcontract Management
- **Software Quality Assurance**
- **Software Configuration Management**

Goals for Key Process Areas in Level 2

- **Software Quality Assurance**
  - ❍ Compliance of the software product and software process with applicable standards, procedures, and product requirements is independently confirmed.
  - ❍ When there are compliance problems, management is aware of them.
  - ❍ Senior management addresses noncompliance issues.
- **Software Configuration Management**
  - ❍ Controlled and stable baselines are established for planning, managing, and building the system.
  - ❍ The integrity of the system's configuration is controlled over time.
  - ❍ The status and content of the software baselines are known.

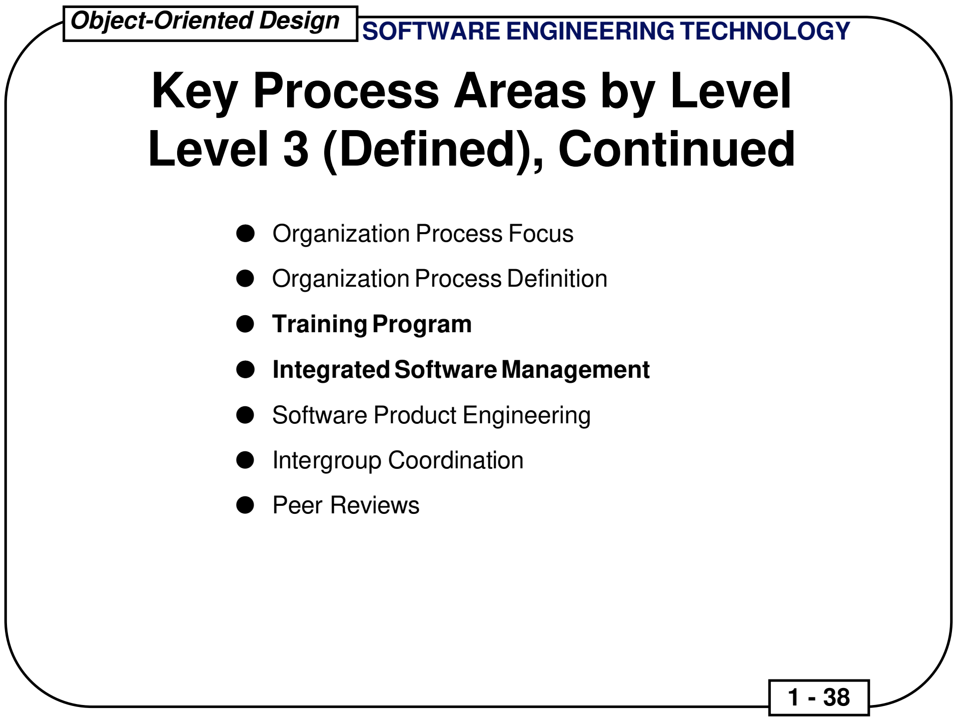


Key Process Areas by Level Level 3 (Defined)

- **Organization Process Focus**
- **Organization Process Definition**
- Training Program
- Integrated Software Management
- Software Product Engineering
- Intergroup Coordination
- Peer Reviews

Goals for Key Process Areas in Level 3

- **Organization Process Focus**
  - ❍ Current strengths and weaknesses of the organization's software process are understood and plans are established to systematically address the weaknesses.
  - ❍ A group is established with appropriate knowledge, skills, and resources to define a standard software process for the organization.
  - ❍ The organization provides the resources and support needed to record and analyze the use of the organization's standard software process in order to maintain and improve it.
- **Organization Process Definition**
  - ❍ A standard software process for the organization is defined and maintained as a basis for stabilizing, analyzing, and improving the performance of the software projects.
  - ❍ Specifications of common software processes and documented process experiences from past and current projects are collected and available.

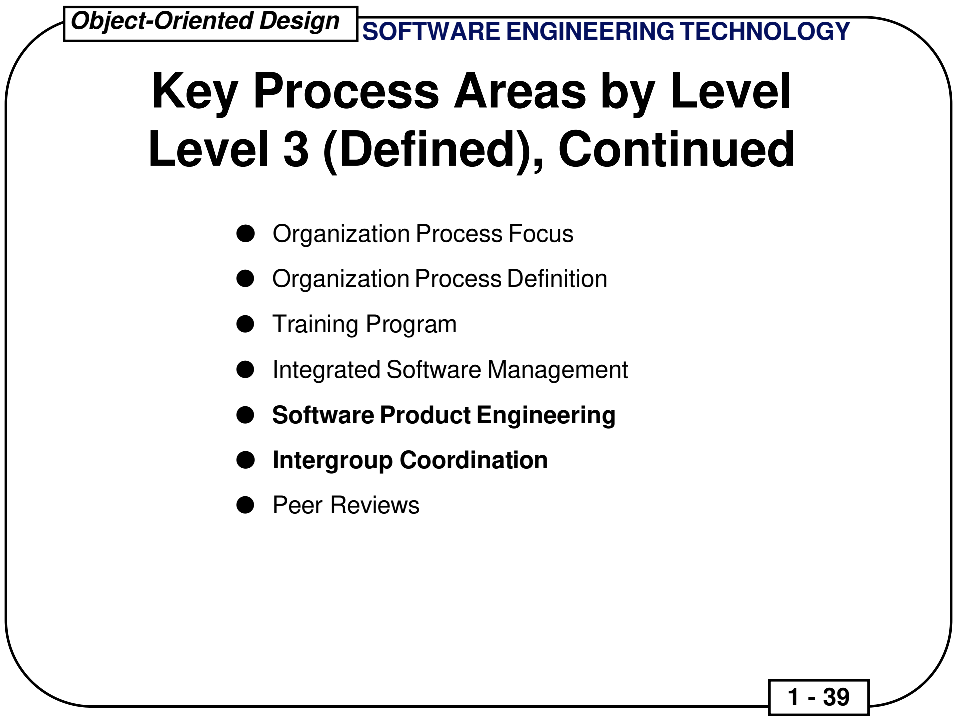


Key Process Areas by Level
Level 3 (Defined), Continued

- Organization Process Focus
- Organization Process Definition
- **Training Program**
- **Integrated Software Management**
- Software Product Engineering
- Intergroup Coordination
- Peer Reviews

Goals for Key Process Areas in Level 3

- **Training Program**
  - The staff and managers have the skills and knowledge to perform their jobs.
  - The staff and managers effectively use, or are prepared to use, the capabilities and features of the existing and planned work environment.
  - The staff and managers are provided with opportunities to improve their professional skills.
- **Integrated Software Management**
  - The planning and managing of each software project is based on the organization's standard software process.
  - Technical and management data from past and current projects are available and used to effectively and efficiently estimate, plan, track, and replan the software projects.

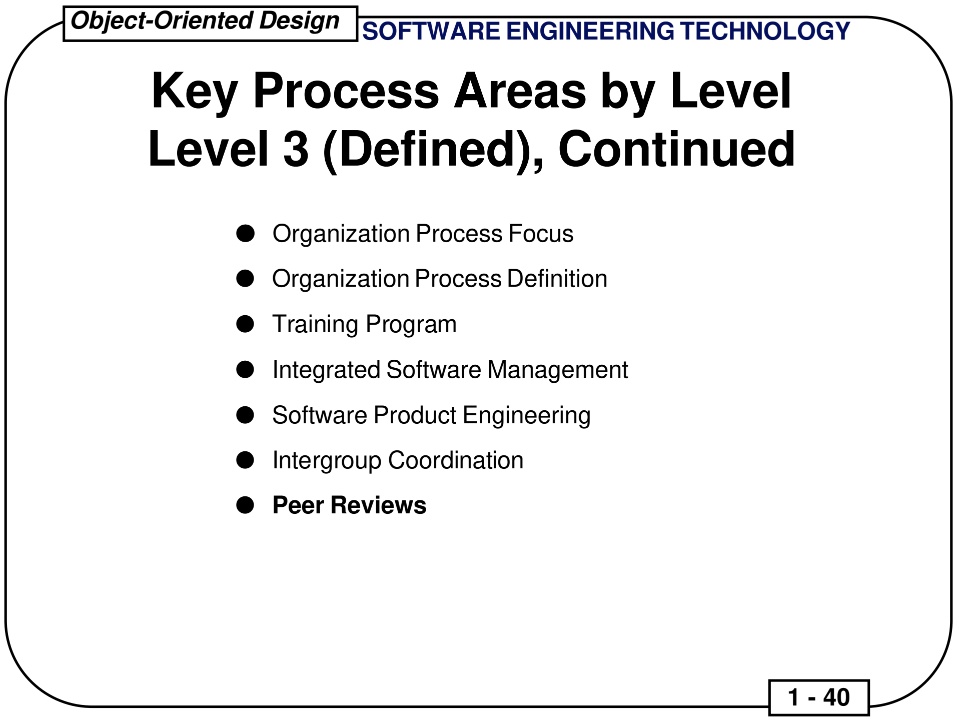


Key Process Areas by Level Level 3 (Defined), Continued

- Organization Process Focus
- Organization Process Definition
- Training Program
- Integrated Software Management
- **Software Product Engineering**
- **Intergroup Coordination**
- Peer Reviews

Goals for Key Process Areas in Level 3

- **Software Product Engineering**
  - ❍ Software engineering issues for the product and the process are properly addressed in the system requirements and system design.
  - ❍ The software engineering activities are well-defined, integrated, and used consistently to produce a software system.
  - ❍ State-of-the-practice software engineering tools and methods are used, as appropriate, to build and maintain the software system.
- **Intergroup Coordination**
  - ❍ The project's technical goals and objectives are understood and agreed to by its staff and managers.
  - ❍ The responsibilities assigned to each of the project groups and the working interfaces between these groups are known to all groups.
  - ❍ The project groups are appropriately involved in intergroup activities and in identifying, tracking, and addressing itnergroup issues.
  - ❍ The project groups work as a team.

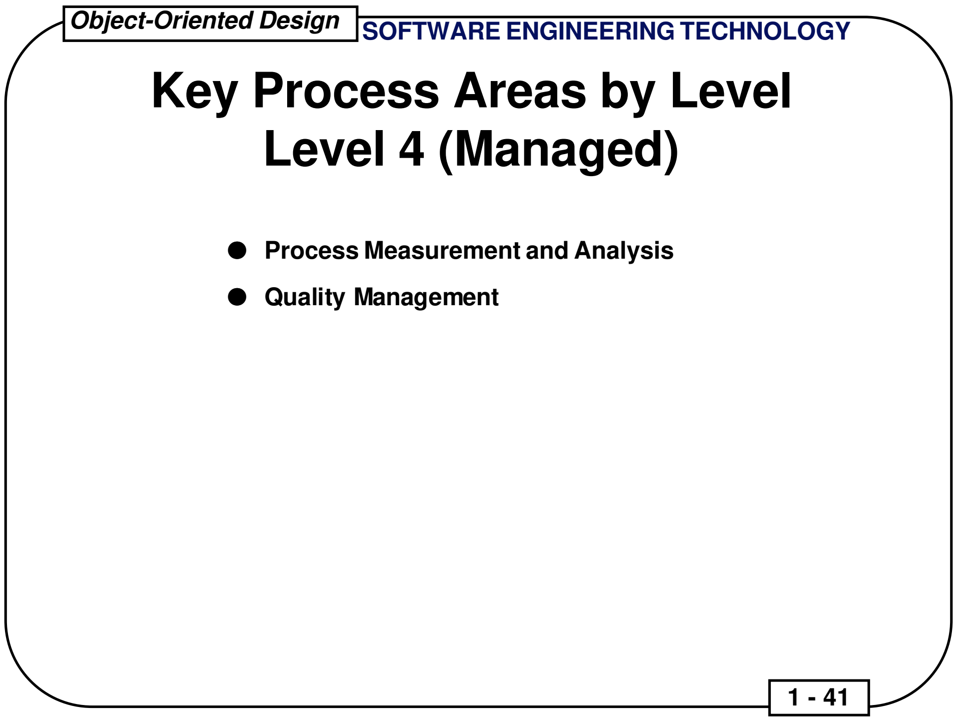


Key Process Areas by Level
Level 3 (Defined), Continued

- Organization Process Focus
- Organization Process Definition
- Training Program
- Integrated Software Management
- Software Product Engineering
- Intergroup Coordination
- **Peer Reviews**

Goals for Key Process Areas in Level 3

- **Peer Reviews**
  - ❍ Product defects are identified and fixed early in the life cycle.
  - ❍ Appropriate product improvements are identified and implemented early in the life cycle.
  - ❍ The staff members become more effective through a better understanding of their work products and knowledge of errors that can be prevented.
  - ❍ A rigorous group process for reviewing and evaluating product quality is established and used.

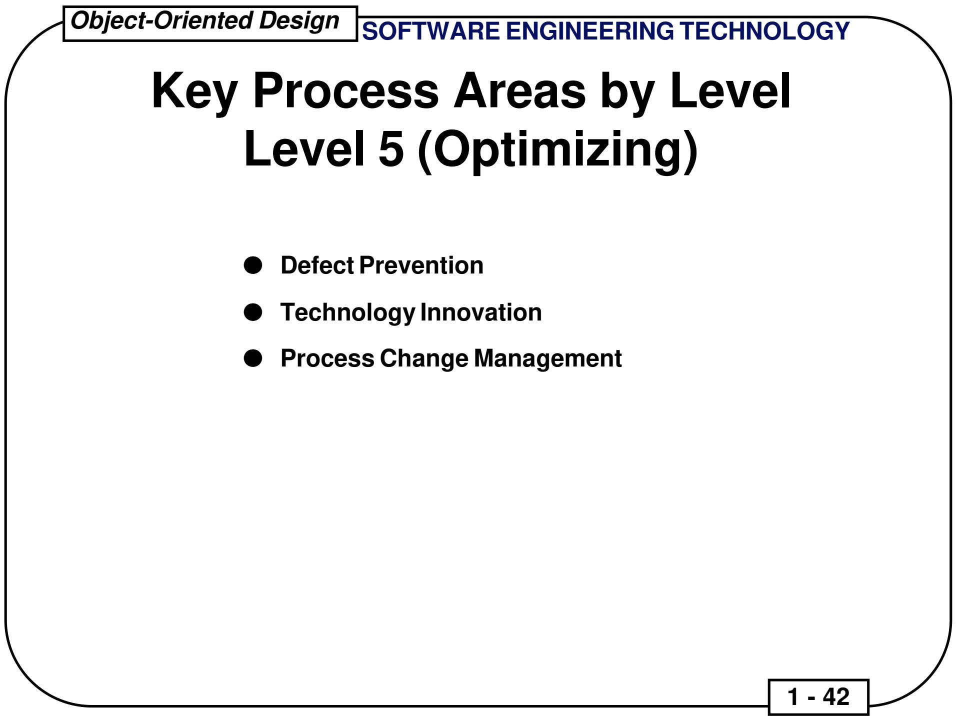


Key Process Areas by Level Level 4 (Managed)

- Process Measurement and Analysis
- Quality Management

Goals for Key Process Areas in Level 4

- **Process Measurement and Analysis**
  - ❍ The organization's standard software process is stable and under statistical process control.
  - ❍ The relationship between product quality, productivity, and product development cycle time is understood in quantitative terms.
  - ❍ Special causes of process variation (i.e., variations attributable to specific applications of the process and not inherent in the process) are identified and controlled.
- **Quality Management**
  - ❍ Measurable goals and priorities for product quality are established and maintained for each software project through interaction with the customer, end users, and project groups.
  - ❍ Measurable goals for process quality are established for all groups involved in the software process.
  - ❍ The software plans, design, and process ar adjusted to bring forecasted process and product quality in line with the goals.
  - ❍ Process measurements are used to manage the software project quantitatively.



Key Process Areas by Level
Level 5 (Optimizing)

- Defect Prevention
- Technology Innovation
- Process Change Management

Goals for Key Process Areas in Level 5

- **Defect Prevention**
  - ❍ Sources of product defects that are inherent or repeatedly occur in the software process activities are identified and eliminated.
- **Technology Innovation**
  - ❍ The organization has a software process and technology capability to allow it to develop or capitalize on the best available technologies in the industry.
  - ❍ Selection and transfer of new technology into the organization is orderly and thorough.
  - ❍ Technology innovations are tied to quality and productivity improvements of the organization's standard software process.
- **Process Change Management**
  - ❍ The organization's staff and managers are actively involved in setting quantitative, measurable improvements goals and in improving the software process.
  - ❍ The organization's standard software process and the projects' defined software processes continually improve.
  - ❍ The organization's staff and managers are able to use the evolving software processes and their supporting tools and methods properly and effectively.

Ada Technology

- *Ada* is a computer programming language specifically designed to support software engineering.
- Some of Ada's features include:
  - ❍ All of the normal constructs for looping, branching, flow control, and subprogram construction
  - ❍ Support for enumeration types, integers, floating point, fixed point, characters, strings, arrays, records, and user-defined data types
  - ❍ Support for algorithm templates (called generics) which allow algorithms to be expressed without concern for the kind of data on which the algorithm is applied
  - ❍ Support for interrupts and concurrent processing
  - ❍ Support for low-level control, such as memory allocation
- Ada is a *design* language as well as a *programming* language.
- Ada is designed to be read by Ada programmers and non-programmers.

Ada

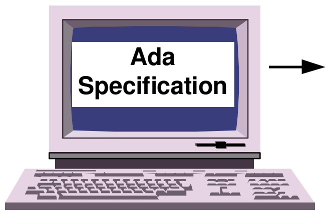
The Ada programming language is defined in detail in:

ANSI/MIL-STD-1815A, **Ada Programming Language**, 22 January 1983, United States Department of Defense, Under Secretary for Defense, Research, and Engineering

To further understand the reasoning behind the choices made in the design of Ada, the following document is highly recommended:

Ichbiah, Barnes, Firth, Woodger, **Rationale for the Design of the Ada Programming Language**, 1986, Honeywell Systems and Research Center, MN65-2100, 3660 Technology Drive, Minneapolis, MN 55418 and Alsys, 29 Avenue de Versailles, 78170 La Celle Saint Cloud, France

Ada Technology



```
with System;
package Sensor is
  type Device is private;
  -- Abstract concept of a sensor
  procedure Define (S : in out Device;
    Where : in System.Address);
  -- Associate a sensor with memory
  function Read(S : in Device)
       return Integer;
  -- Return sensed value
private
  -- details omitted
end Sensor;
```

Ada Code Example

This is an example of an Ada package which defines a class of objects of type **Sensor.Device.**

This code example is incomplete. The details of the private section of the package specification are omitted, and the package body, in which the procedure **Define** and the function **Read** are implemented, is not shown.

Ada Technology

- **From the software engineering perspective, Ada helps by acting as something much more than a programming language; Ada can be used as a common language for communicating:**
  - ❍ **Some aspects of the requirements**
  - ❍ **Some aspects of the design**
  - ❍ **All aspects of the code**
- **In particular, by using Ada as a *design language,* code is simply realized as a complete, detailed elaboration of a design.**
- **For large, multi-person teams, Ada can be used as an exact, precise way to communicate requirements and design information -- often in a form which may be syntactically checked by a compiler. Ada is much better than conventional English in this regard.**

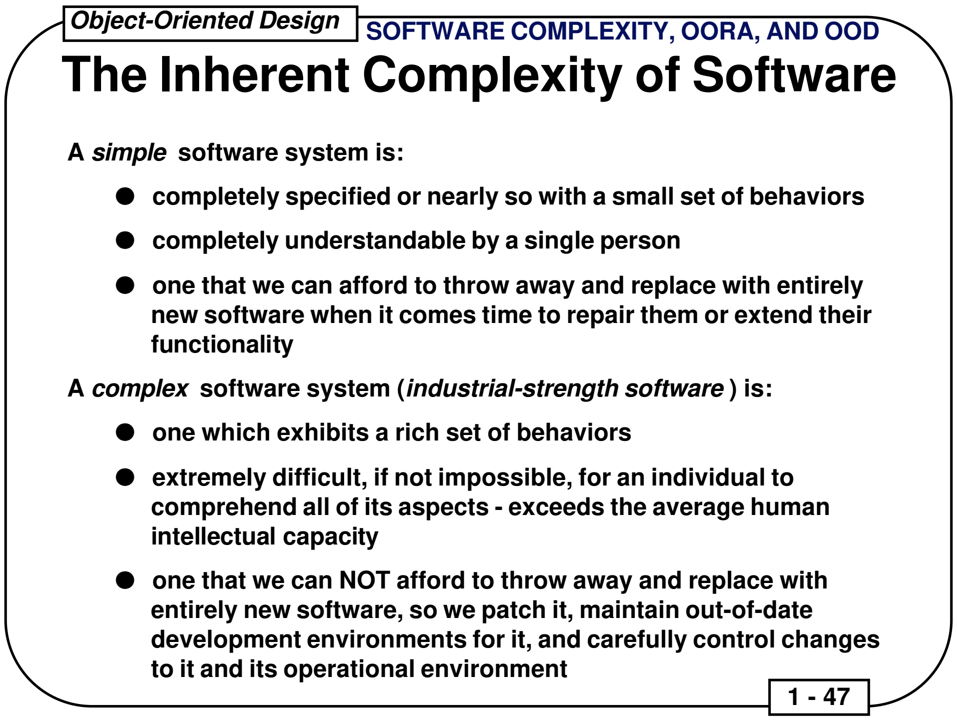
Suggested Reading

- Grady Booch, **Object Oriented Design with Applications**, 1991, The Benjamin/Cummings Publishing Company, Inc., ISBN 0-8053-0091-0
- Grady Booch, **Software Components with Ada**, 1987, The Benjamin/Cummings Publishing Company, Inc., ISBN 0-8053-0610-2
- Grady Booch, **Software Engineering with Ada**, 1987, The Benjamin/Cummings Publishing Company, Inc., ISBN 0-8053-0604-8
- R.J.A. Buhr, **System Design with Ada**, 1984, Prentice-Hall, Inc., ISBN 0-13-881623-9
- David Naiditch, **Rendezvous with Ada: A Programmer's Introduction**, 1989, John Wiley & Sons, ISBN 0-471-61654-0
- The Software Productivity Consortium, **Ada Quality and Style: Guidelines for Professional Programmers**, 1989, Van Nostrand Reinhold, ISBN 0-442-23805-3

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
- ✓ Bringing Order to Chaos
- ✓ On Designing Complex Systems

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



Object-Oriented Design

SOFTWARE COMPLEXITY, OORA, AND 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**

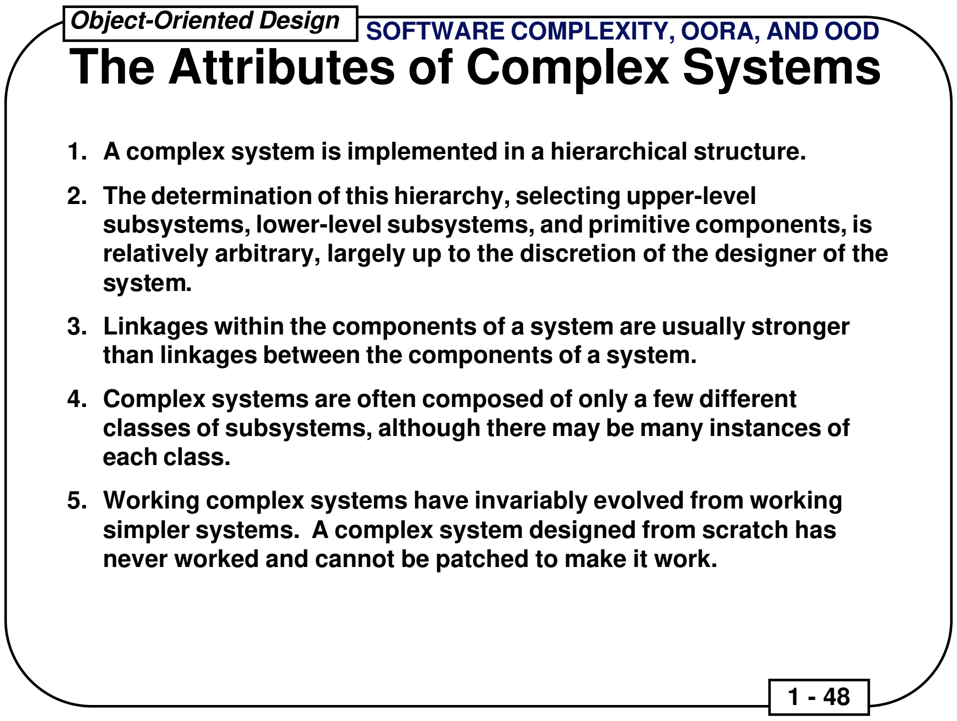
1 - 47

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

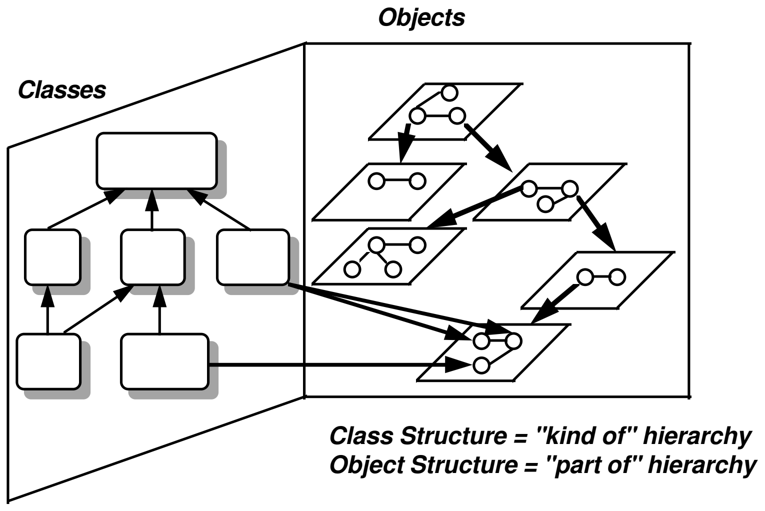
1. **A complex system is implemented in a hierarchical structure.**
2. **The determination of this hierarchy, selecting upper-level subsystems, lower-level subsystems, and primitive components, is relatively arbitrary, largely up to the discretion of the designer of the system.**
3. **Linkages within 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

narly 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

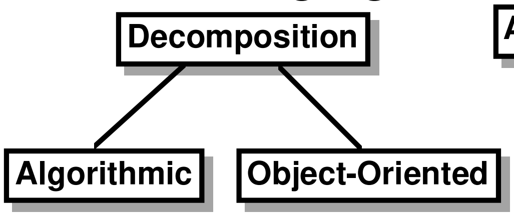
Most complex systems do not embody a single hierarchy. Complex systems are usually composed of a network of related hierarchies. Two broad types of hierarchies exist:

- 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

Bringing Order to Chaos



Abstraction and Hierarchy

The Role of Decomposition

- "Divide and Conquer"
- Break a large system into a number of smaller subsystems and so on until the subsystems are manageable and understandable
- Two views: algorithmic and object-oriented

The Role of Abstraction

- Reducing the number of concepts to comprehend concurrently
- 7 +/- 2

The Role of Hierarchy

- Understanding the relationships between entities (object structure)
- Understanding the redundancy in the entities (class structure)

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**