# Overview

<table>
<thead>
<tr>
<th>Lecture</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>What is Software Engineering?</td>
</tr>
<tr>
<td>2</td>
<td>Software Processes and Life Cycles</td>
</tr>
<tr>
<td>3</td>
<td>Planning and Managing a Project</td>
</tr>
<tr>
<td>4</td>
<td>Capturing Requirements</td>
</tr>
<tr>
<td>5</td>
<td>Designing the System</td>
</tr>
<tr>
<td>6</td>
<td>UML and Patterns</td>
</tr>
<tr>
<td>7</td>
<td>Concerning Objects</td>
</tr>
<tr>
<td>8</td>
<td>Testing the Program</td>
</tr>
<tr>
<td>9</td>
<td>Testing the System</td>
</tr>
<tr>
<td>10</td>
<td>OO – Encapsulation, Consinace, Type Conformance</td>
</tr>
<tr>
<td>11</td>
<td>Introduction to the CMM</td>
</tr>
<tr>
<td>12</td>
<td>Introduction to the CMM (continued.)</td>
</tr>
<tr>
<td>13</td>
<td>Delivery and Maintenance</td>
</tr>
<tr>
<td>14</td>
<td>Evaluating Process and The Future of Software Engineering</td>
</tr>
</tbody>
</table>
What Is Software?

- Software is what makes a general purpose device special purpose.
  - A pre-recorded video tape is software for a VCR.
  - More commonly, we tend to think of software in terms of computers.

Definition of Software

- Software is a computer program, along with associated data and documentation, that is intended to be used by persons other than the author.
Software Engineering

• Design, implementation, and maintenance of software.

Importance of Software

• The economies of ALL developed nations are dependent on software
• More and more systems are software controlled
• Software engineering expenditure represents a significant fraction of GNP in all developed countries
Software Costs

- Software costs often dominate system costs. The costs of software on a PC are often greater than the hardware cost.
- Software costs more to maintain than it does to develop. For systems with a long life, maintenance costs may be several times development costs.

What Is a Process?

- The text defines process as “a series of steps involving activities, constraints, and resources that produce an intended output of some kind.”
- ISO/IEC 12207 defines process as “a set of interrelated activities, which transform inputs into outputs.”
- Mark Paulk (author of the CMM® for Software) gives the definition “what you do to do what you do.”

® Capability Maturity Model and CMM are registered trademarks of Carnegie Mellon University.
Process Characteristics

- Prescribes all of the major activities.
- Uses resources subject to a set of constraints, and produces intermediate and final products.
- May be composed of sub-processes that are linked together or as a hierarchy.
- Each activity has an entrance and exit criteria.
- Activities are organized in a sequence.
- There are a set of guiding principals that explain the goals of each activity.
- Constraints may be applied to an activity, resource, or product.

What Is a Life Cycle?

- In biology the term *life cycle* is “the continuous sequence of changes undergone by an organism from one primary form to the development of the same form again*.”
  - An insect starts an egg, which becomes a larva, then a pupa, and finally an adult, which lays eggs, starting the cycle over again.
- A software product undergoes a sequence of changes starting with an initial concept and ending with its retirement.

* The Random House Dictionary of the English Language, 1966
Stages in Software Development

• Requirements analysis and definition
• System design
• Program design
• Writing the program
• Unit testing
• Integration testing
• System testing
• System delivery
• Maintenance

Life Cycle Models

• Code and Fix
• Waterfall Model
• ‘V’ Model
• Prototyping Model
• Operational Specification
• Transformational Model
• Phased Development
• Spiral Model
• Unified Development Model
Unified Process Life Cycle

Why Model Processes?

• Whatever you do, you use a process to do it.
• By modeling the process, you can gain understanding so that:
  – The process is repeatable
  – The process is manageable
  – The process can be improved
Terminology Confusion

There are many different views of the Software [Life Cycle] Process[es].

• ISO/IEC 12207
• IEEE 1074
• SEI Capability Maturity Model® for Software

© Capability Maturity Model and CMM are registered trademarks of Carnegie Mellon University.

ISO/IEC 12207

Standard for Software Life Cycle Processes

• Primary Life Cycle Processes:
  – Acquisition, Supply, Development, Operation, Maintenance
• Supporting Live Cycle Processes:
  – Documentation, Configuration Management, Quality Assurance, Verification, Validation, Joint Review, Audit
• Organizational Life Cycle Processes:
  – Management, Infrastructure, Improvement, Training
IEEE 1074

Standard for Developing
Software Life Cycle Processes

• Activity Groupings:
  – Project Management
  – Pre-Development
  – Development
  – Post-Development
  – Integral

CMM® for Software

Defines Key Process areas by Level

• Level 2
  – Requirements Management, Project Planning, Project Tracking and Oversight, Subcontract Management, Quality Assurance, Configuration Management

• Level 3

• Level 4
  – Quantitative Process Management, Software Quality Management

• Level 5
  – Defect Prevention, Technology Change Management, Process Change Management

© Capability Maturity Model and CMM are registered trademarks of Carnegie Mellon University.
Common Theme

• 12207 requires a supplier to “define or select a software life cycle model” and map the processes, activities, and tasks of 12207 to that model.
• IEEE 1074 requires organizing and mapping the activities into a Software Life Cycle Process.
• The CMM, at Level 3, requires an Organization to define an Organizational Software Process, and then for each project tailor it to form the Project’s Software Defined Process.

Customer Questions

• Do you understand customer problem and needs?
• Can you design a system to solve customer problem or satisfy customer needs?
• How long will it take you to develop the system?
• How much will it cost to develop the system?
Project Deliverables

- Documents
- Demonstrations of function
- Demonstrations of subsystems
- Demonstrations of accuracy
- Demonstrations of reliability, performance or security

Milestones and Activities

- Activity: takes place over a period of time
- Milestone: completion of an activity -- a particular point in time
- Precursor: event or set of events that must occur in order for an activity to start
- Duration: length of time needed to complete an activity
- Due date: date by which an activity must be completed
Slack or Float Time

Slack time = available time - real time

= latest start time - earliest start time

Earned Value Management

Tracks both cost and schedule in a unified manner.

- Each activity has an estimated cost.
- When an activity is scheduled to be complete, add that activity’s estimated cost to the total called Budgeted Cost of Work Scheduled (BCWS).
- When an activity is in fact complete, add that activity’s estimated cost to the total called Budgeted Cost of Work Performed (BCWP).
- The cost expended to date is the Actual Cost of Work Performed (ACWP).
Cost and Schedule Variance

• These indicate current project status
  – Cost variance:
    • CV = BCWP – ACWP
    • Positive indicates costs less than expected
  – Schedule variance:
    • SV = BCWP – BCWS
    • Positive indicates ahead of schedule

Cost and Schedule Indices

• These can be applied (with caution) to the work remaining to get a predicted final cost and end data
  – Cost Index
    • CI = BCWP/ACWP
  – Schedule Index
    • SI = BCWP/BCWS
Cautions About Earned Value

- More effective if there are several milestones due in each reporting period.
- If an activity crosses a reporting point, 50% credit to BCWS and to BCWP if the activity has started. This reduces a false indication of behind schedule and overrun. Note it can provide a false indication of being on schedule or underrun.
- Do not compare BCWS and ACWP. This merely indicates that the spending is to plan, but does not consider the progress toward the goal.

Capturing the Requirements

- Requirement: a feature of the system or a description of something the system is capable of doing in order to fulfill the system’s purpose
- Three kinds of requirements:
  - those that absolutely must be met
  - those that are highly desirable but not necessary
  - those that are possible but could be eliminated
Why are Requirements Important?

• Causes of failed projects:
  – Incomplete requirements
  – Lack of user involvement
  – Lack of resources
  – Unrealistic expectations
  – Lack of executive support
  – Changing requirements
  – Lack of planning
  – System no longer needed

Testable Requirements

• Requirement
  – “Water quality information must be accessible immediately.”

• What does immediately mean?

• Testable Requirement
  – “Water quality information must be accessible within 5 seconds of request.”
Making Requirements Testable

- Specify a quantitative description for each adverb and adjective so that the meaning of the quantifiers is clear and unambiguous.
- Replace pronouns with specific names of entities.
- Make sure that every noun is defined in exactly one place in the requirements documents.

Requirements Documents

- Requirements definition: complete listing of what the customer expects the system to do
- Requirements specification: restates the definition in technical terms so that the designer can start on the design
- Configuration management: supports direct correspondence between the two documents
**Configuration Management**

- Set of procedures that track
  - requirements that define what the system should do
  - design modules that are generated from requirements
  - program code that implements the design
  - tests that verify the functionality of the system
  - documents that describe the system

**Functional Vs. Non-functional Requirements**

- **Functional**: describes an interaction between the system and its environment
- **Examples**:
  - System shall communicate with external system X.
  - What conditions must be met for a message to be sent

- **Non-functional**: describes a restriction or constraint that limits our choices for constructing a solution
- **Examples**:
  - Paychecks distributed no more than 4 hours after initial data are read.
  - System limits access to senior managers.
Types of Requirements

- Physical environment
- Interfaces
- Users and human factors
- Functionality
- Documentation
- Data
- Resources
- Security
- Quality assurance

Characteristics of Requirements

- Are they correct?
- Are they consistent?
- Are they complete?
- Are they realistic?
- Does each describe something the customer needs?
- Are they verifiable?
- Are they traceable?
Static Descriptions of Requirements

• Indirect reference
  – Example: k equations in n unknowns
• Recurrence relations
  – Example: F(0)=1; F(1)=1; F(n+1)=F(n)+F(n-1)
• Axiomatic definition
• Expression as a language
  – Example: Backus-Naur form

Object-oriented Specifications

• Each entity in the system is an object.
• A method or operation is an action that can be performed directly by the object or can happen to the object.
• Encapsulation: the methods form a protective boundary around an object.
• Class hierarchies of objects encourage inheritance.
• Polymorphism: same method for different objects, each with different behavior
Additional Notations

• Hierarchical techniques
  – Warnier diagrams
• Data flow diagrams
• Software Requirements Engineering Methodology (SREM)
• Structured Analysis and Design Technique (SADT)
• Z (pronounced Zed)

Prototyping requirements

• Throw-away prototypes
• Evolutionary prototypes
• Rapid prototypes
**Requirements documentation**

- Requirements definition document: what the customer wants
  - general purpose
  - background and objectives of system
  - description of customer-suggested approach
  - detailed characteristics
  - operational environment
- Requirements specification document: what the designers need to know

**Participants in requirements process**

- Contract monitors
- Customers and users
- Business managers
- Designers
- Testers
- Requirements analysts
Requirements review

• Review goals and objections of system.
• Compare requirements with goals and objectives.
• Describe operational environment.
• Examine
  – interfaces
  – information flow
  – functions
• Check for omissions, incompleteness, inconsistency.
• Document risk.
• Discuss how system will be tested.

Choosing a specification technique

• Applicability
• Implementability
• Testability/simulation
• Checkability
• Maintainability
• Modularity
• Level of abstraction/expressibility
• Soundness
• Verifiability
• Run-time safety
• Tools maturity
• Looseness
• Learning curve
• Technique maturity
• Data modeling
• Discipline
Conceptual Design

• Tells the customer what the system will do
• Answers:
  – Where will the data come from?
  – What will happen to the data in the system?
  – What will the system look like to users?
  – What choices will be offered to users?
  – What is the timing of events?
  – What will the reports and screens look like?

Characteristics of Good Conceptual Design

• in customer language with no technical jargon
• describes system functions
• independent of implementation
• linked to requirements
# Technical Design

- Tells the programmers what the system will do
- Includes:
  - major hardware components and their function
  - hierarchy and function of software components
  - data structures
  - data flow

# Five Ways to Create Designs

- Modular decomposition
- Data-oriented decomposition
- Event-oriented decomposition
- Outside-in design
- Object-oriented design
Three Design Levels

- Architecture: associates system components with capabilities
- Code design: specifies algorithms and data structures for each component
- Executable design: lowest level of design, including memory allocation, data formats, bit patterns

Design Styles

- Pipes and filters
- Object-oriented design
- Implicit invocation
- Layering
- Repositories
- Interpreters
- Process control
- Client-server
Important Design Issues

- Modularity and levels of abstraction
- Collaborative design
- Designing the user interface
  - metaphors, mental model, navigation rules, look and feel
  - cultural issues
  - user preferences
- Concurrency
- Design patterns and reuse

Causes of Design Breakdown

- Lack of specialized design schemes
- Lack of meta-scheme about design process leading to poor allocation of resources
- Poor prioritization of issues leading to poor selection of alternative solutions
- Difficulty in considering all the stated or inferred constraints
- Difficulty in performing mental simulations with many steps or test cases
- Difficulty in keeping track and returning to subproblems whose solution had been postponed
- Difficulty in merging solutions from individual subproblems to form a complete solution
Cultural and Language Issues

- Individuals from a culture where harmony is important, may not express individual opinions.
- Using electronic communication, nuances expressed by gestures and facial expressions are lost.
- Translation from one natural language to another is not precise.
  - There are over 500 words to describe pasta in Italian
  - There are over 160 words for “camel” in Bedoin

Cultural Issues

- Must consider beliefs, values, norms, traditions, mores, and myths of those who will use the system.
- Different numerical formats.
- Text descriptions in various languages have varying lengths.
- Meanings of colors.
  - England: purple represents royalty
  - Japan: purple represents dignity and nobility
  - Greece: death and evil
Concurrency

- **Mutual exclusion**
  - A component desiring to change the state of an object:
    - Locks the object.
    - Changes the state.
    - Unlocks the object.

- **Monitors**
  - An abstract object (in the OO sense of the word)
    - Mutator methods perform the locking and unlocking.

- **Guardians**
  - A task that is always running to control access to a resource.
    - Uses a rendezvous to coordinate multiple requests.

Characteristics of Good Design

- **Component independence**
  - coupling
  - cohesion

- **Exception identification and handling**

- **Fault prevention and tolerance**
  - active
  - passive
Content coupling
Common coupling
Control coupling
Stamp coupling
Data coupling
Uncoupled

Functional
Sequential
Communication
Procedural
Temporal
Logical
Coincidental
Object orientation

- identity
- abstraction
- classification
- encapsulation
- inheritance
- polymorphism
- persistence

Objects and classes

- Every object has a name (also called a reference or handle).
- Objects can have attributes (such as color, size, location).
- Objects can have operations or behaviors (such as takeoff, land, repair).
- Each object is an instance of a class.
- A specific implementation of an operation for a certain class is called a method.
OO design

- Usually uses an OO requirements representation
- System design identifies and represents objects and classes, plus details of each objects attributes and behaviors.
- System design also identifies interactions and relationships.
- Program design inserts computational features in the models.
- Program design inserts class library details.
- Program design considers nonfunctional requirements to enhance design.

Use cases

- Diagrams have four elements:
  - actors
  - cases
  - extensions
  - uses
Identifying participants

- What users or groups use the system to perform a task?
- What users or groups are needed so that the system can perform its functions?
- What external systems use the system to perform a task?
- What external systems, users or groups send information to the system?
- What external systems, users or groups receive information from the system?

UML and the OO process

- Workflow diagrams
- Object model
- Sequence diagrams
- Collaboration diagrams
- Package diagrams
- Component diagrams
- Deployment diagrams
First cut at object classes

- Structures
- External systems
- Devices
- Roles
- Operating procedures
- Places
- Organizations
- Things that are manipulated by the system to be built

Guidelines for building classes

- What needs to be “processed” in some way?
- What items have multiple attributes?
- When do you have more than one object in a class?
- What is based on the requirements themselves, not derived from your understanding of the requirements?
- What attributes and operations are always applicable to a class or object?
Guidelines for identifying behaviors

- Imperative verbs
- Passive verbs
- Actions
- Things or reminded events
- Roles
- Operating procedures
- Services provided by an organization

How does software fail?

- Wrong requirement: not what the customer wants
- Missing requirement
- Requirement impossible to implement
- Faulty design
- Faulty code
- Improperly implemented design
Terminology

- Fault identification: what fault caused the failure?
- Fault correction: change the system
- Fault removal: take out the fault

Types of faults

- Algorithmic fault
- Syntax fault
- Computation and precision fault
- Documentation fault
- Stress or overload fault
- Capacity or boundary fault
- Timing or coordination fault
- Throughput or performance fault
- Recovery fault
- Hardware and system software fault
- Documentation fault
- Standards and procedures fault
Testing Issues

- Test organization
- Attitudes toward testing
- Who performs the tests?
- Views of the test objectives

Test Organization

- Module, Component, or Unit Testing
- Integration testing
- Function testing
- Performance testing
- Acceptance testing
- Installation testing
Attitudes Toward Testing

- New programmers not accustomed to viewing testing as a discovery process.
- Testing, by the student, generally designed to demonstrate that the program works under certain circumstances.
- May consider the presence of faults as a critique of ability.
- Customers are not interested in programs that work under certain circumstances, they want programs that work under all circumstances.

Egoless Programming

- Programs are viewed as components of a larger system.
- Programs are not the property of those who wrote them.
- When a fault is discovered or a failure occurs, the egoless development team is concerned with correcting the fault, not with placing the blame.
White Box Example

- The flowchart shows a program that executes $n \times m$ times.
- Would not want to pick large values of $n$ and $m$.
- Consider the cases where
  - $n > m$
  - $n = m$
  - $n < m$

Examining the Code

- Code Walkthroughs
  - Code and accompanying documentation is presented by the author to the review team.
  - Author leads and controls the discussion.
  - Atmosphere is informal.
- Code Inspections
  - Review team checks the code against a prepared list of concerns (checklist).
  - Several steps involved – team members generally review code individually and reconvene.
  - Trained moderator leads and controls the discussion.
  - Atmosphere is formal.
Proving code correct

- Formal proof techniques
- Symbolic execution
- Automated theorem-proving

Formal Proof

Based upon notation and concepts developed by C. A. R. Hoare.

\[ \text{P} \{Q\} \text{R} \]

If the proposition \( P \) is true, execution of the program \( Q \) will result in the proposition \( R \) being true.

An alternative notation (adapted for C and C++)

\[ /* P */ Q /* R */ \]

or

//Precondition
(block of statements)
//Postcondition
**Axiom of Assignment**

Let $P$ be some proposition containing the symbol $x$.
Let $e$ be some expression. (Note, the evaluation of $e$ is assumed to be without side-effects.)

$P_0$ is the proposition $P$ in which every occurrence of $x$ is replaced by $e$.

Note this is **not** every occurrence of $e$ in $P_0$ replaced by $x$.

Or using our C notation

```c
/* P0 */
x = f;
/* P */
```

**Rules of Consequence**

A rule of inference has this general form:

If $X$ and $Y$ then $Z$  

which may be interpreted

If $X$ has been proven (or is an axiom)  
and $Y$ has been proven (or is an axiom)  
then, $Z$ is considered proven.

If $P \land (Q \Rightarrow R)$ then $P \land Q \Rightarrow R$  
If $P \land (Q \Rightarrow S)$ then $P \land S \Rightarrow Q$  
If $P \land (Q \Rightarrow S)$ then $Q \land P \Rightarrow S$  
If $P \land (Q \Rightarrow S)$ then $Q \land S \Rightarrow P$  
If $P \land (Q \Rightarrow S)$ then $Q \land S \Rightarrow P$
Rule of Composition

Programs generally consist of more than one statement.
Using our C notation:

If
/* P */ Q1 /* R */) and /* R1 */ Q2 /* R */
then /* P */ (Q1 Q2) /* R */

Rule of Iteration

Let P be a proposition whose truth is not changed by the execution of the statement S.
Let B be a proposition whose truth is necessary for the continued execution of the loop.
In the C notation
If
/* P && B */
S
/* P */
then /* P */
while (B) S
/* !B && P */
Advantages and Disadvantages

- Formal proofs can discover algorithmic faults in the code.
- Regular use forces rigorous and precise specification.
- More successful when the program and the proof are developed simultaneously.
- Much work is involved in setting up and carrying out the proof.
- Proofs are not always correct. The history of mathematics is full of proofs that were later shown to be fallacious.

Test thoroughness

- Statement testing
  - Every statement is executed at least once.
- Branch testing
  - At every decision point, each branch is chosen at least once.
- Path testing
  - Every path through the code is executed at least once.
- Definition-use testing
  - Every path from every definition of every variable to every use of that definition is exercised.
Test Thoroughness

- All-uses testing
  - The test set includes at least one path from every definition to every use that can be reached by that definition.

- All-predicate-uses/some-computational-uses
  - For every variable and every definition of that variable, a test includes at least one path from the definition to every predicate use.

- All-computational-uses/some-predicate-uses
  - For every variable and every definition of that variable, a test includes at least one path from the definition to every computational use.

Principals of System Testing

- Unit and integration test is intended to ensure that the code implements the design.
- I.E., the programmers wrote the code to do what the designers intended.
- System testing is to ensure that the system does what the customer wants.
System testing process

- Function testing: does the integrated system perform as promised by the requirements specification?
- Performance testing: are the non-functional requirements met?
- Acceptance testing: is the system what the customer expects?
- Installation testing: does the system run at the customer site(s)?

Techniques used in system testing

- Build or spin plan for gradual testing
- Configuration management
  - versions and releases
  - production system vs. development system
  - deltas, separate files and conditional compilation
  - change control
- Regression testing
Reliability, Availability, and Maintainability

- **Reliability:**
  - System operates as expected over a long period of time.
- **Availability:**
  - System operates when you need it.
- **Maintainability:**
  - System can be fixed (or modified) efficiently and effectively.

Failure Severity

- *Catastrophic*: a failure may cause death of system loss.
- *Critical*: a failure that may cause severe injury or major system damage that results in mission loss.
- *Marginal*: a failure that may cause minor injury or minor system damage that results in delay, loss of availability, or mission degradation.
- *Minor*: a failure not serious enough to cause injury or system damage, but that results in unscheduled maintenance or repair.
Measuring Reliability, Availability, and Maintainability

- Average of the interfailure times is called the Mean Time To Failures (MTTF).
- Average of the repair times is called Mean Time to Repair (MTTR).
- The Mean Time Between Failures (MTBF) is the sum MTTF + MTTR.
- Reliability: $R = \frac{MTTF}{1+MTTF}$
- Availability: $A = \frac{MTBF}{1+MTBF}$
- Maintainability: $M = \frac{1}{1+MTTR}$

Testing safety-critical systems

- Design diversity: use different kinds of designs, designers
- Software safety cases: make explicit the ways the software addresses possible problems
  - failure modes and effects analysis
  - hazard and operability studies
- Cleanroom: certifying software with respect to the specification
Design Diversity

• Concept:
  – Different teams design to the same requirements (functional and performance).
  – Different source languages.
  – Different computer architectures.

• Result:
  – Teams tend to make common mistakes anyway.
  – Synchronization problems.

Users of a System

• Users
  – Exercise main system functions
  – Use the system to solve customer’s problems

• Operator
  – Perform supplemental functions
User Training

• Major System Functions
• Need not be aware of system internal operation.
• Relate to old system
  – Psychological barrier to learning new system

Operator Training

• Support Functions
• How the system works, not what it does
  – Start-up/Shut-down
  – Configuration
  – Grant/Deny access
  – Assign resources (e.g., disk space)
  – Monitor & tune system performance
  – Recover lost files
Configuration control process

- Problem discovered by or change requested by user/customer/developer, and recorded
- Change reported to the configuration control board
- CCB discusses problem: determines nature of change, who should pay
- CCB discusses source of problem, scope of change, time to fix; they assign severity/priority and analyst to fix
- Analyst makes change on test copy
- Analyst works with librarian to control installation of change
- Analyst files change report

Introduction
To The
The Software Engineering Institute’s
Capability Maturity Model®
for Software
Capability Maturity Model (CMM)

A conceptual framework to help organizations:

- Characterize the maturity of their process (As Is)
- Establish goals for process improvement (To Be)
- Set priorities for immediate actions (Transition)
- Manage and sustain change in organizations (Stabilize)
- Introduce changes incrementally, to avoid disrupting current processes

Immature Software organizations

Processes are ad hoc, and occasionally chaotic

Product quality is unpredictable

Costs and schedules are usually exceeded

Testing and reviews usually curtailed under stress

Success rides on individual talent
Mature Software Organizations

Processes are defined and documented
Management plans, monitors, and communicates
Roles and responsibilities are clear
Product and process are measured
Quality, cost and schedules are predictable

SEI Process Maturity Levels

- **Initial (1)**: Basic Management Control
- **Repeatable (2)**: Process Architecture
- **Defined (3)**: Measure & Control
- **Managed (4)**: CQI
- **Optimizing (5)**: Continuously Improving Process
Level 1 - The initial Process

Unstable environment lacking project management disciplines

Practitioners argue against engineering discipline under the guise of "Individual creativity"

Standard and practices often sacrificed to schedule

Process capability is unpredictable

Schedule, cost, and quality target are rarely met

Success rides on individual talent and heroic effort

Level 2 - The Repeatable Process

Basic project management discipline is installed to ensure that software engineering practices are followed and measured.

Previously successful processes are repeatable in stable environment

Reasonable commitments are planned

Process capability exists for meeting business needs and schedules

Project metrics are collected

Foundation for continuous quality improvement is in place
Level 3 - The Defined Process

Common software processes are defined, documented, and applied throughout the organization

Process of sharing successful practices is in place

Roles and responsibilities are well defined

Organization processes are measured

Process capability exists to meet schedule, cost, and functionality target within established product lines

Organization metrics are collected and used for CQI

Level 4 - The Managed Process

Statistical process control principles are used to address special causes of process variation

Process measurement and product measurement are available and correlated for "fine tuning" of existing processes

Standard software processes are established based on facts and data collected on software engineering practices

Process capability exists to perform within narrowly defined quantitative limits - Targets are predictable.

Business enterprise metrics are collected and used for CQI
Level 5 - The Optimizing Process

Software process is continually improved

Chronic causes of poor performance are eliminated

New technologies are prototyped, piloted, and if successful, introduced into existing process

Process capability is continually raised

Key Concept

Maturity levels can not be skipped because each level forms the necessary foundation from which to achieve the next level.

A level 1 organization that tries to implement a defined process (level 3) before it has established a repeatable process (level 2) is usually unsuccessful because project managers are overwhelmed by schedule and cost pressures.

It is suggested that the organization focuses on management processes before engineering processes.

Although it seems easier to define and implement an engineering process than a management process, without management discipline, the engineering process is sacrificed to schedule and cost pressures.