# UNIT V

**Product Metrics:** A Frame work for Product Metrics, Metrics for the Requirements Model, Metrics for the Design Model, Metrics for Testing, Metrics for Maintenance.
**Estimation:** Software Project Estimation, Decomposition Techniques, Empirical Estimation Models, Specialized Estimation Techniques.
**Software Configuration Management:** Software Configuration Management. **Software Process Improvement**: The SPI Process, The CMMI.

## Product Metrics:

### A Frame work for Product Metrics

***SOFTWARE QUALITY***

> **Software quality** is defined as the conformance to explicitly stated functional and performance requirements, explicitly documented development standards, and implicit characteristics that are expected of all professionally developed software

**McCall's Quality Factors:** Factors that affect software quality can be categorized in two broad groups:

1. Factors that can be directly measured (e.g. defects uncovered during testing)
2. Factors that can be measured only indirectly (e.g. usability or maintainability)

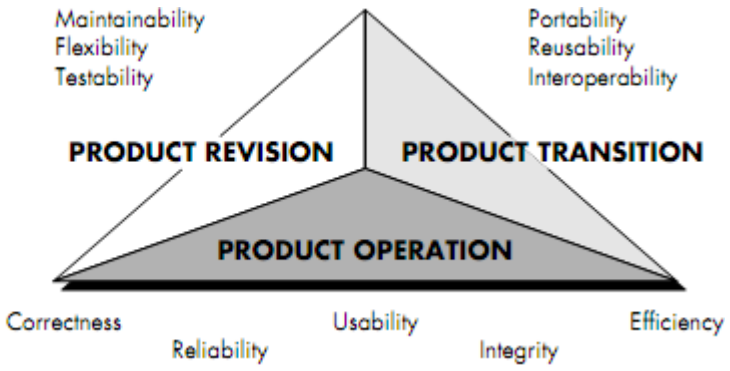



**FIGURE 19.1** McCall's software quality factors

Figure 19.1; focus on three important aspects of a software product: its operational characteristics, its ability to undergo change, and its adaptability to new environments.

McCall and his colleagues provide the following descriptions:

1. **Correctness:** The extent to which a program satisfies its specification and fulfills the customer's mission objectives.
2. **Reliability:** The extent to which a program can be expected to perform its intended function with required precision.
3. **Efficiency:** The amount of computing resources and code required by a program to perform its function.
4. **Integrity:** Extent to which access to software or data by unauthorized persons can be controlled.
5. **Usability:** Effort required to learn, operate, prepare input, and interpret output of a program.
6. **Maintainability:** Effort required to locate and fix an error in a program.
7. **Flexibility:** Effort required to modify an operational program.
8. **Testability:** Effort required to test a program to ensure that it performs its intended function.
9. **Portability:** Effort required to transfer the program from one hardware and/or software system environment to another.
10. **Reusability:** Extent to which a program [or parts of a program] can be reused in other applications related to the packaging and scope of the functions that the program performs.
11. **Interoperability:** Effort required to couple one system to another.

## ISO 9126 Quality Factors

The ISO 9126 standard was developed in an attempt to identify the key quality attributes for computer software. The standard identifies six key quality attributes**:**

1. **Functionality:** The degree to which the software satisfies stated needs as indicated by the following sub attributes: *suitability, accuracy, interoperability, compliance, and security.*
2. **Reliability:** The amount of time that the software is available for use as indicated by the following sub attributes: *maturity, fault tolerance, recoverability.*
3. **Usability:** The degree to which the software is easy to use as indicated by the following sub attributes: *understandability, learnability, operability.*
4. **Efficiency:** The degree to which the software makes optimal use of system resources as indicated by the following sub attributes: *time behavior, resource behavior.*
5. **Maintainability:** The ease with which repair may be made to the software as

indicated by the following sub attributes: *analyzability, changeability, stability, testability.*

6. **Portability:** The ease with which the software can be transposed from one environment to another as indicated by the following sub attributes: *adaptability, installability, conformance, replaceability.*

***Measures, Metrics and Indicators***

- **A *measure*** provides a quantitative indication of the extent, amount, dimension, capacity, or size of some attribute of a product or process
- **The IEEE glossary defines a *metric*** as "a quantitative measure of the degree to which a system, component, or process possesses a given attribute."
- **An *indicator*** is a metric or combination of metrics that provide insight into the software process, a software project, or the product itself

**METRICS FOR ANALYSIS MODEL**

These metrics examine the analysis model with the intent of predicting the "size" of the resultant system. Size is an indicator of design complexity and is almost always an indicator of increased coding, integration and testing effort.

**Function-Based Metrics:** The function-point metric can be used effectively as a means for measuring the functionality delivered by the system. Using historical data FP metric can be used to:

1) Estimate cost or effort required to design code and test the software.
2) Predict number of errors that will be encountered during testing

3) Forecast number of components and number of projected source lines in the implemented system. Function points are derived using an empirical relationship based on countable measures of software information domain.

**Number of External inputs (EI)**: Each external input originates from a user or is transmitted from another application. Inputs are often used to update Internal Logic Files.

**Number of External Outputs**: Each external output is derived data within the application that provides information to the user. External outputs refer to reports, screens, error messages

**Number of external inquiries:** An external inquiry is defined as an online input that results in generation of some intermediate software response in

form of an online output

**Number of internal logical files:** Each internal logical file is a logical grouping of data that resides within the applications boundary and is maintained via external inputs.

**Number of external interface files:** Each external interface file is a logical grouping of data that resides external to application but provides information

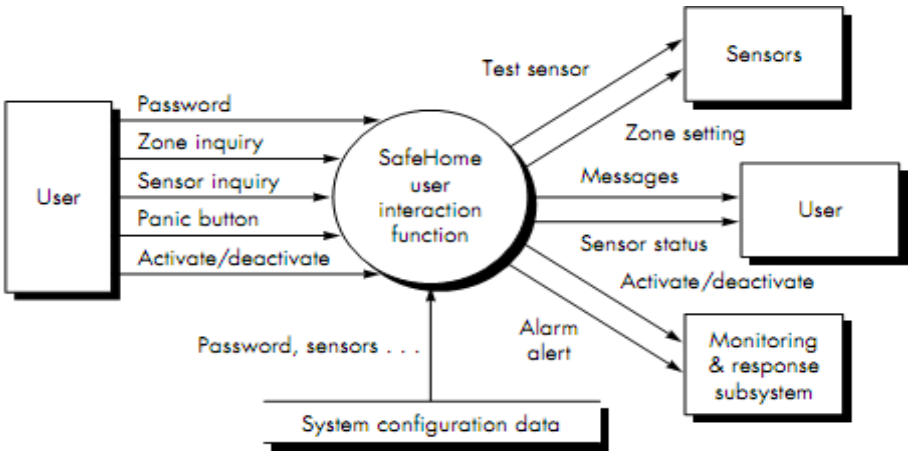



**FIGURE 19.3** Part of the analysis model for SafeHome software

that may be of use to application

| | | | Weighting Factor | | | |
|---|---|---|---|---|---|---|
| **Measurement parameter** | **Count** | | **Simple** | **Average** | **Complex** | |
| Number of user inputs | 3 | × | 3 | 4 | 6 | = 9 |
| Number of user outputs | 2 | × | 4 | 5 | 7 | = 8 |
| Number of user inquiries | 2 | × | 3 | 4 | 6 | = 6 |
| Number of files | 1 | × | 7 | 10 | 15 | = 7 |
| Number of external interfaces | 4 | × | 5 | 7 | 10 | = 20 |
| Count total | | | | | | 50 |

**FIGURE 19.4** Computing function points for a SafeHome function

Three user inputs—password, panic button, and activate/deactivate—are shown in the figure along with two inquires—zone inquiry and sensor inquiry. One file (system configuration file) is shown. Two user outputs (messages and sensor status) and four external interfaces (test sensor, zone setting, activate/deactivate, and alarm alert) are also present. These data, along with the appropriate complexity, are shown in Figure 19.4. The count total shown in Figure 19.4 must be adjusted using Equation

$$FP = count\ total * [0.65 + 0.01 * \sum(Fi)]$$

where count total is the sum of all FP entries obtained from Figure 19.3 and

Fi (i = 1to 14) are "complexity adjustment values." For the purposes of this example, we assume that (∑Fi) is 46 (a moderately complex product). Therefore,

***FP = 50 * [0.65 + 0.01 *46] = 56***

## Metrics For Specification Of Quality

List of characteristics that can be used to assess the quality of the analysis model and the corresponding requirements specification: specificity (lack of ambiguity), completeness, correctness, understandability, verifiability, internal and external consistency, achievability, concision, traceability, modifiability, precision, and reusability.

We assume that there are Nr requirements in a specification, such that

$$n_r = n_f + n_{nf}$$

Where nf is the number of functional requirements and Nnf is the number of non-functional (e.g., performance) requirements.

To determine the **specificity** (lack of ambiguity) of requirements

$$Q_1 = n_{ui}/n_r$$

$n_{ui}$ is the number of requirements for which all reviewers had identical interpretations. The closer the value of Q to 1, the lower is the ambiguity of the specification.

The **completeness** of functional requirements can be determined by computing the ratio

$$Q_2 = n_u/[n_i \times n_s]$$

$n_u$ is the number of unique function requirements, $n_i$, is the number of inputs (stimuli) defined or implied by the specification, and **ns** is the number of states specified. The Q2 ratio measures the percentage of necessary functions that have been specified for a system

**METRICS FOR DESIGN MODEL**

**Architectural Design Metrics:** Architectural design metrics focus on

characteristics of the program architecture. These metrics are black box in the sense that they do not require any knowledge of the inner workings of a particular software component.

Card and Glass define ***three*** software ***design complexity measures***: Structural complexity, Data complexity, and System complexity.

**Structural complexity** of a module i is defined in the following manner:

$$S(i) = f^2 out(i)$$

where fout(i) is the fan-out of module i.(Fan-out means number of modules directly sub-ordinate to module i)

**Data complexity** provides an indication of the complexity in the internal interface for a module i and is defined as

$$D(i) = v(i)/[fout(i) + 1]$$

where v(i) is the number of input and output variables that are passed to and from module i.

**System complexity** is defined as the sum of structural and data complexity, specified as

$$C(i) = S(i) + D(i)$$

As each of these complexity values increases, the overall architectural complexity of the system also increases. This leads to a greater likelihood that integration and testing effort will also increase.

**Morphology (shape) metrics:** It is a function of the number of modules and the number of interfaces between modules **size = n + a**

where n is the number of nodes and a is the number of arcs.

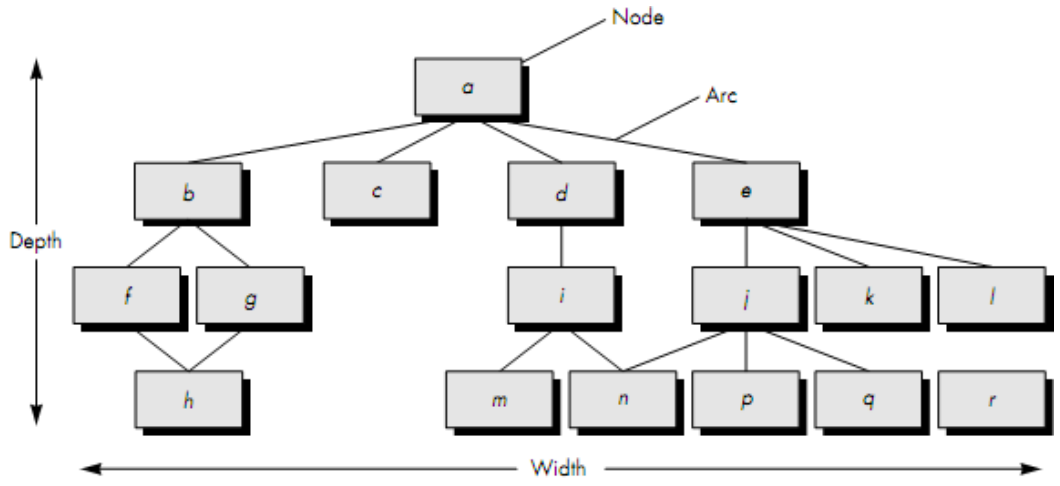



**FIGURE 19.5** Morphology metrics

For the architecture shown in Figure 19.5,

**size** = 17 + 18 = 35

**depth** = the longest path from the root (top) node to a leaf node. For the architecture shown in Figure 19.5, depth = 4.

**width** = maximum number of nodes at any one level of the architecture. For the architecture shown in Figure 19.5, width = 6.arc-to-node ratio, r = a/n, r = 18/17 = 1.06.

**DSQI (Design Structure Quality Index):** US air force has designed the DSQI. Compute s1 to s7 from data and architectural design

- S1: Total number of modules
- S2: Number of modules whose correct function depends on the data input
- S3: Number of modules whose function depends on prior processing
- S4: Number of data base items
- S5: Number of unique database items
- S6: Number of database segments
- S7: Number of modules with single entry and exit

***Calculate D1 to D6 from s1 to s7 as follows:***

- D1=1 if standard design is followed otherwise D1=0
- D2(module independence)=(1-(s2/s1))
- D3(module not depending on prior processing)=(1-(s3/s1))
- D4(Data base size)=(1-(s5/s4))
- D5(Database compartmentalization)=(1-(s6/s4)
- D6(Module entry/exit characteristics)=(1-(s7/s1))

$$DSQI = \sum WiDi$$

where i = 1 to 6, wi is the relative weighting of the importance of each of the intermediate values, and
$\sum wi = 1$ (if all Di are weighted equally, then wi= 0.167).

DSQI of present design be compared with past DSQI. If DSQI is significantly lower than the average, further design work and review are indicated

**Metrics For Object-Oriented Design:** Whitmire [WHI97] describes nine distinct and measurable characteristics of an OO design:

**Size:** Size is defined in terms of four views: population, volume, length, and functionality

**Complexity:** How classes of an OO design are interrelated to one another

**Coupling:** The physical connections between elements of the OO design

**Sufficiency:** "the degree to which an abstraction possesses the features required of it, or the degree to which a design component possesses features in its abstraction, from the point of view of the current application."

**Completeness:** An indirect implication about the degree to which the abstraction or design component can be reused.

**Cohesion:** The degree to which all operations working together to achieve a single, well-defined purpose.

**Primitiveness:** Applied to operations and classes, the degree to which an operation is atomic.

**Similarity:** The degree to which two or more classes are similar in terms of their structure, function, behavior, or purpose.

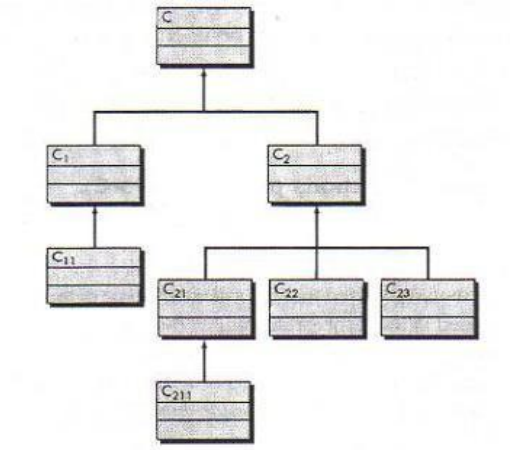
**Volatility:** Measures the likelihood that a change will occur.

**Class-Oriented Metrics-The CK Metrics suite:** Proposed by Chidamber and Kemerer

**Weighted methods per class:** Assume that n methods of complexity c1,c2,---cn are defines for a class C WMC=$\sum$ci for i=1 ton

The number of methods and their complexity are reasonable indicators of amount or effort required to implement and test a class.

**Depth of the inheritance tree:** Max length form node to root of tree referring to fig DIT=4. As DIT grows low-level classes will inherit many methods, this leads to many difficulties & greater design complexity.




***Number of children****:* The subclasses that are immediately subordinate to a class in class hierarchy are termed its children. As NOC grows reuse increases but abstraction represented by parent class is diluted if some children are not appropriate members of parent class.

***Coupling between object classes:*** As CBO increases reusability decreases

***Response for a class:*** A set of methods that can potentially be executed in response to a message.RFC is no. of methods in response set. As RFC increases complexity increases.

***Lack of cohesion in methods:*** LCOM is no. of methods that access one or more of same attributes. If no methods access same attribute LCOM=0

**Class-Oriented Metrics: The MOOD Metrics Suite**

*Method Inheritance Factor:*

$$\text{MIF} = \Sigma\, M_i(C_i) / \Sigma\, M_a(C_i)$$

where the summation occurs over $i = 1$ to $T_c$. $T_c$ is defined as the total number of classes in the architecture; $C_i$ is a class within the architecture; and

$$M_a(C_i) = M_d(C_i) + M_i(C_i)$$

where

$M_a(C_i)$ = the number of methods that can be invoked in association with $C_i$.
$M_d(C_i)$ = the number of methods declared in the class $C_i$.
$M_i(C_i)$ = the number of methods inherited (and not overridden) in $C_i$.

***Coupling factor:***

$$CF = \Sigma_i \Sigma_j \, is\_client\,(C_i, C_j)/(T_c^2 - T_c)$$

where the summations occur over $i = 1$ to $T_C$ and $j = 1$ to $T_C$. The function

$is\_client = 1$, if and only if a relationship exists between the client class, $C_c$, and the server class, $C_s$, and $C_c \neq C_s$

$= 0$, otherwise

If CF increases the complexity of OO software also increases.

**Class-Oriented Metrics Proposed by Lorenz and Kidd**

Lorenz and Kidd divide class-based metrics into four broad categories

- **Size-Oreinted Metrics**: focus on count of attributes and operations for an individual class *Inheritance-Based Metrics:* focus on manner in which operations are reused through class hierarchy *Merics For Class Internal:* focus on cohesion

*Metrics For External:* focus on coupling

- ***Component-Level design metrics***

*Cohesion metrics:* a function of data objects and the focus of their definition

*Coupling metrics:* a function of input and output parameters, global variables, and modules.

$d_o$ = number of output data parameters
$c_o$ = number of output control parameters

For global coupling,

$g_d$ = number of global variables used as data
$g_c$ = number of global variables used as control

For environmental coupling,

$w$ = number of modules called (fan-out)
$r$ = number of modules calling the module under consideration (fan-in)

Using these measures, a module coupling indicator, $m_c$, is defined in the following way:

$$m_c = k/M$$

where $k$ is a proportionality constant and

$$M = d_i + (a \times c_i) + d_o + (b \times c_o) + g_d + (c \times g_c) + w + r \qquad (15\text{-}6)$$

Values for $k$, $a$, $b$, and $c$ must be derived empirically.

As the value of $m_c$ increases, the overall module coupling decreases. In order to have the coupling metric move upward as the degree of coupling increases, a revised coupling metric may be defined as

$$C = 1 - m_c$$

*Complexity metrics:* hundreds have been proposed (e.g., Cyclomatic complexity)

- ***Operation-Oriented Metrics***
    - ✔ average operation size
    - ✔ operation complexity
    - ✔ average number of parameters per operation
- ***Interface Design Metrics***

  *Layout appropriateness:* a function of layout entities, the geographic position and the "cost" of making transitions among entities. This is a worthwhile design metric for interface design.

**METRICS FOR SOURCE CODE**

- Primitive measure that may be derived after the code is generated or estimated once design is complete. Length $N = n_1 \log_2 n_1 + n_2 \log_2 n_2$ Program volume $V = N \log_2 (n_1 + n_2)$ Volume ratio $L=2/n1 * n2/N2$ Where $n_1$ = the number of distinct operators that appear in a program $n_2$ = the number of distinct operands that appear in a program $N_1$ = the total number of operator occurrences.
  $N_2$ = the total number of operand occurrence.

**METRICS FOR TESTING**

- Program Level and Effort
- $PL = 1/[(n_1 / 2) \times (N_2 / n_2 l)]$
- $e = V/PL$

**METRICS FOR MAINTENANCE**

IEEE standard suggests a software maturity index that provides indication of stability of software product. The Software Maturity Index, SMI, is defined as:

$$\mathbf{SMI = [M_t - (F_c + F_a + F_d)/ M_t]}$$

Where $M_t$ = the number of modules in the current release

$F_c$ = the number of modules in the current release

that have been changed $F_a$ = the number of

modules in the current release that have been

added.

$F_d$ = the number of modules from the preceding release that were deleted in the current release

## Estimation:

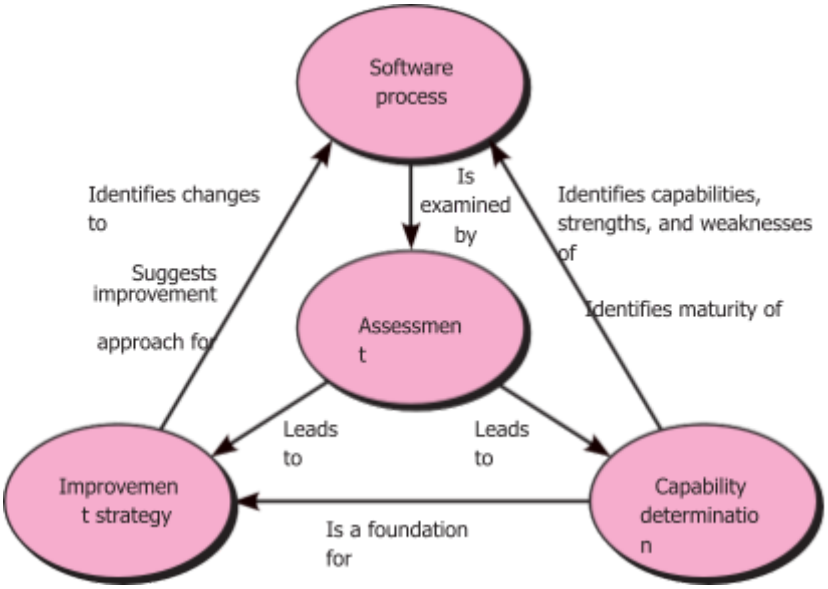
# Software Process Improvement:

The term *software process improvement* (SPI) implies many things. First, it implies that elements of an effective software process can be defined in an effective manner; second, that an existing organizational approach to software development can be assessed against those elements; and third, that a meaningful strategy for improvement can be defined. The SPI strategy transforms the existing approach to software development into something that is more focused, more repeatable, and more reliable (in terms of the quality of the product produced and the timeliness of delivery).

Because SPI is not free, it must deliver a return on investment. The effort and time that is required to implement an SPI strategy must pay for itself in some measurable way. To do this, the results of improved process and practice must lead to a reduction in software "problems" that cost time and money. It must reduce the number of defects that are delivered to end users, reduce the amount of rework due to quality problems, reduce the costs associated with software maintenance and support, and reduce the indirect costs that occur when software is delivered late.

### Approaches to SPI

Although an organization can choose a relatively informal approach to SPI, the vast majority choose one of a number of SPI frameworks. An *SPI framework* defines (1) a set of characteristics that must be present if an effective software process is to be achieved, (2) a method for assessing whether those characteristics are present, (3) a mechanism for summarizing the results of any assessment, and (4) a strategy for assisting a software organization in implementing those process characteristics that have been found to be weak or missing.




An SPI framework assesses the "maturity" of an organization's software process and provides a qualitative indication of a maturity level. In fact, the term "maturity model" (Section 30.1.2) is often applied. In essence, the SPI framework encompasses a maturity model that in turn incorporates a set of process quality indicators that pro- vide an overall measure of the process quality that will lead to product quality.

Figure 30.1 provides an overview of a typical SPI framework. The key elements of the framework and their relationship to one another are shown.

You should note that there is no universal SPI framework. In fact, the SPI frame- work that is chosen by an organization reflects the constituency that is championing the SPI effort. Conradi [Con96] defines six different SPI support constituencies:

**Quality certifiers.** Process improvement efforts championed by this group focus on the following relationship:

$$\textbf{Quality}(\textit{Process}) \Rightarrow \textbf{Quality}(\textit{Product})$$

Their approach is to emphasize assessment methods and to examine a

well- defined set of characteristics that allow them to determine whether the process exhibits quality. They are most likely to adopt a process framework such as the CMM, SPICE, TickIT, or Bootstrap.[1]

**Formalists.** This group wants to understand (and when possible, optimize) process workflow. To accomplish this, they use process modeling languages (PMLs) to create a model of the existing process and then design extensions or modifications that will make the process more effective.

**Tool advocates.** This group insists on a tool-assisted approach to SPI that models workflow and other process characteristics in a manner that can be analyzed for improvement.

**Practitioners.** This constituency uses a pragmatic approach, "emphasizing mainstream project-, quality- and product management, applying project-level planning and metrics, but with little formal process modeling or enactment support" [Con96].

**Reformers.** The goal of this group is organizational change that might lead to a better software process. They tend to focus more on human issues (Section 30.5) and emphasize measures of human capability and structure.

**Ideologists.** This group focuses on the suitability of a particular process model for a specific application domain or organizational structure. Rather than typical software process models (e.g., iterative models), ideologists would have a greater interest in a process that would, say, support reuse or reengineering.

As an SPI framework is applied, the sponsoring constituency (regardless of its over- all focus) must establish mechanisms to: (1) support technology transition, (2) deter- mine the degree to which an organization is ready to absorb process changes that are proposed, and (3) measure the degree to which changes have been adopted.

## Maturity Models

A *maturity model* is applied within the context of an SPI framework. The intent of the maturity model is to provide an overall indication of the "process maturity" exhibited by a software organization. That is, an indication of the quality of the software process, the degree to which practitioner's understand and apply the process, and the general state of software engineering practice. This is accomplished using some type of ordinal scale.

[1] For example, the Software Engineering Institute's *Capability Maturity Model*

five levels of maturity [Sch96]:

***Level 5, Optimized***—The organization has quantitative feedback systems in place to identify process weaknesses and strengthen them pro-actively. Project teams analyze defects to determine their causes; software processes are evaluated and updated to pre- vent known types of defects from recurring.

***Level 4, Managed***—Detailed software process and product quality metrics establish the quantitative evaluation foundation. Meaningful variations in process performance can be distinguished from random noise, and trends in process and product qualities can be predicted.

***Level 3, Defined***—Processes for management and engineering are documented, standardized, and integrated into a standard software process for the organization.

All projects use an approved, tailored version of the organization's standard software process for developing software.

***Level 2, Repeatable***—Basic project management processes are established to track cost, schedule, and functionality. Planning and managing new products is based on experience with similar projects.

***Level 1, Initial***—Few processes are defined, and success depends more on individ- ual heroic efforts than on following a process and using a synergistic team effort.

The CMM maturity scale goes no further, but experience indicates that many organ- izations exhibit levels of "process immaturity" [Sch96] that undermine any rational attempt at improving software engineering practices. Schorsch [Sch06] suggests four levels of immaturity that are often encountered in the real world of software development organizations:

*Level 0, Negligent*—Failure to allow successful development process to succeed. All prob- lems are perceived to be technical problems. Managerial and quality assurance activities are deemed to be overhead and superfluous to the task of software development process. Reliance on silver pellets.

*Level –1, Obstructive*—Counterproductive processes are imposed. Processes are rigidly defined and adherence to the form is stressed. Ritualistic ceremonies abound. Collective management precludes assigning responsibility. Status quo über alles.

*Level –2, Contemptuous*—Disregard for good software engineering institutionalized. Complete schism between software development activities and software process improvement activities. Complete lack of a training program.

*Level –3, Undermining*—Total neglect of own charter, conscious discrediting of peer organization's software process improvement efforts. Rewarding failure and poor performance.

Schorsch's immaturity levels are toxic for any software organization. If you encounter any one of them, attempts at SPI are doomed to failure.

The overriding question is whether maturity scales, such as the one proposed as part of the CMM, provide any real benefit. I think that they do. A maturity scale provides an easily understood snapshot of process quality that can be used by practitioners and managers as a benchmark from which improvement

strategies can be planned.

**THE CMMI :**

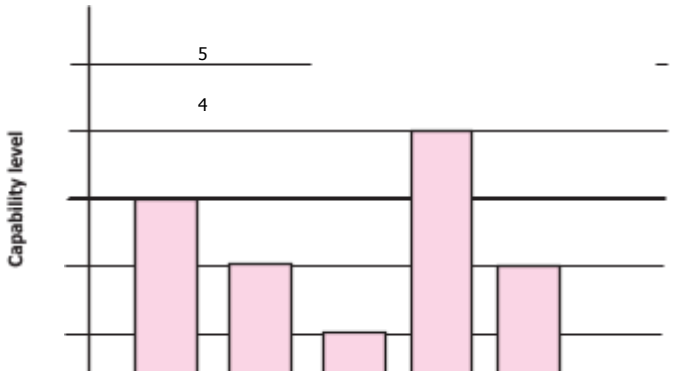
WebRef
Complete information on the IDRE can be obtained at HYPERLINK "http://www.sei.cmu/" "_th www.sei.cmu/ -mbu/cmmi/".

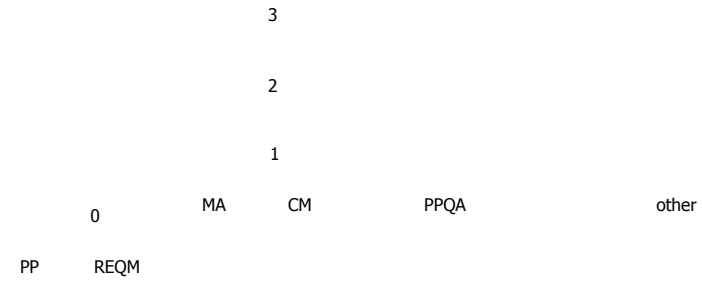
The original CMM was developed and upgraded by the Software Engineering Insti- tute throughout the 1990s as a complete SPI framework. Today, it has evolved into the *Capability Maturity Model Integration* (CMMI) [CMM07], a comprehensive process meta-model that is predicated on a set of system and software engineering capabil- ities that should be present as organizations reach different levels of process capa- bility and maturity.

The CMMI represents a process meta-model in two different ways: (1) as a "continuous" model and (2) as a "staged" model. The continuous CMMI meta- model describes a process in two dimensions as illustrated in Figure 30.2. Each process area (e.g., project planning or requirements management) is formally assessed against specific goals and practices and is rated according to the follow- ing capability levels:

**Level 0:** *Incomplete*—the process area (e.g., requirements management) is either not performed or does not achieve all goals and objectives defined by the CMMI for level 1 capability for the process area.

**Level 1:** *Performed*—all of the specific goals of the process area (as defined by the CMMI) have been satisfied. Work tasks required to produce defined work products are being conducted.




3
2
1
0
MA
CM
PPQA
other
PP
REQM

**Process area**

**Level 2:** *Managed*—all capability level 1 criteria have been satisfied. In addi- tion, all work associated with the process area conforms to an organizationally defined policy; all people doing the work have access to adequate resources to get the job done; stakeholders are actively involved in the process area as required; all work tasks and work products are "monitored, controlled, and reviewed; and are evaluated for adherence to the process description" [CMM07].

**Level 3:** *Defined*—all capability level 2 criteria have been achieved. In addi- tion, the process is "tailored from the organization's set of standard processes according to the organization's tailoring guidelines, and con- tributes work products, measures, and other process-improvement informa- tion to the organizational process assets" [CMM07].

**Level 4:** *Quantitatively managed*—all capability level 3 criteria have been achieved. In addition, the process area is controlled and improved using measurement and quantitative assessment. "Quantitative objectives for qual- ity and process performance are established and used as criteria in managing the process" [CMM07].

**Level 5:** *Optimized*—all capability level 4 criteria have been achieved. In addition, the process area is adapted and optimized using quantitative (statistical) means to meet changing customer needs and to continually improve the efficacy of the process area under consideration.

The CMMI defines each process area in terms of "specific goals" and the "specific practices" required to achieve these goals. *Specific goals* establish the characteristics that must exist if the activities implied by a process area are to be effective. *Specific practices* refine a goal into a set of process-related activities.

For example, **project planning** is one of eight process areas defined by the CMMI for "project management" category.[6] The specific goals (SG) and the associated specific practices (SP) defined for **project planning** are [CMM07]:

**SG 1 Establish Estimates**

SP 1.1-1 Estimate the Scope of the Project

SP 1.2-1 Establish Estimates of Work Product and Task Attributes SP 1.3-1 Define Project Life Cycle

SP 1.4-1 Determine Estimates of Effort and Cost

**SG 2 Develop a Project Plan**

SP 2.1-1 Establish the Budget and Schedule SP 2.2-1 Identify Project Risks

SP 2.3-1 Plan for Data Management SP 2.4-1 Plan for Project Resources

SP 2.5-1 Plan for Needed Knowledge and Skills SP 2.6-1 Plan Stakeholder Involvement

SP 2.7-1 Establish the Project Plan

**SG 3 Obtain Commitment to the Plan**

SP 3.1-1 Review Plans That Affect the Project SP 3.2-1 Reconcile Work and Resource Levels SP 3.3-1 Obtain Plan Commitment

In addition to specific goals and practices, the CMMI also defines a set of five generic goals and related practices for each process area. Each of the five generic goals corresponds to one of the five capability levels. Hence, to achieve a particular capability level, the generic goal for that level and the generic practices that corre- spond to that goal must be achieved. To illustrate, the generic goals (GG) and prac- tices (GP) for the **project planning** process area are [CMM07]:

**GG 1 Achieve Specific Goals**

GP 1.1 Perform Base Practices

**GG 2 Institutionalize a Managed Process** GP 2.1 Establish an Organizational Policy GP 2.2 Plan the Process

GP 2.3 Provide Resources GP 2.4 Assign

Responsibility GP

2.5 Train People

GP 2.6 Manage Configurations

GP 2.7 Identify and Involve Relevant

Stakeholders GP 2.8 Monitor and Control the

Process

GP 2.9 Objectively Evaluate Adherence

GP 2.10 Review Status with Higher-Level Management

**GG 3 Institutionalize a Defined Process**

GP 3.1 Establish a Defined Process

GP 3.2 Collect Improvement Information

**GG 4 Institutionalize a Quantitatively Managed Process**

GP 4.1 Establish Quantitative Objectives for the

Process GP 4.2 Stabilize Subprocess Performance

**GG 5 Institutionalize an Optimizing Process**

GP 5.1 Ensure Continuous Process

Improvement GP 5.2 Correct Root Causes

of Problems

| Level | Focus | Process Areas |
|---|---|---|
| **Optimized** | ***Continuous improvement*** | **Continuous workforce innovation Organizational performance alignment Continuous capability improvement** |
| **Predictable** | ***Quantifies and manages knowledge, skills, and abilities*** | **Mentoring Organizational capability management Quantitative performance management Competency-based assets Empowered workgroups Competency integration** |
| **Defined** | ***Identifies and develops knowledge, skills, and abilities*** | **Participatory culture Workgroup development Competency-based practices Career development Competency development Workforce planning Competency analysis** |
| **Managed** | ***Repeatable, basic people management practices*** | **Compensation Training and development Performance management Work environment Communication and co-ordination Staffing** |
| **Initial** | ***Inconsistent practices*** | |