Capability Maturity Model AND its levels

Maturity model

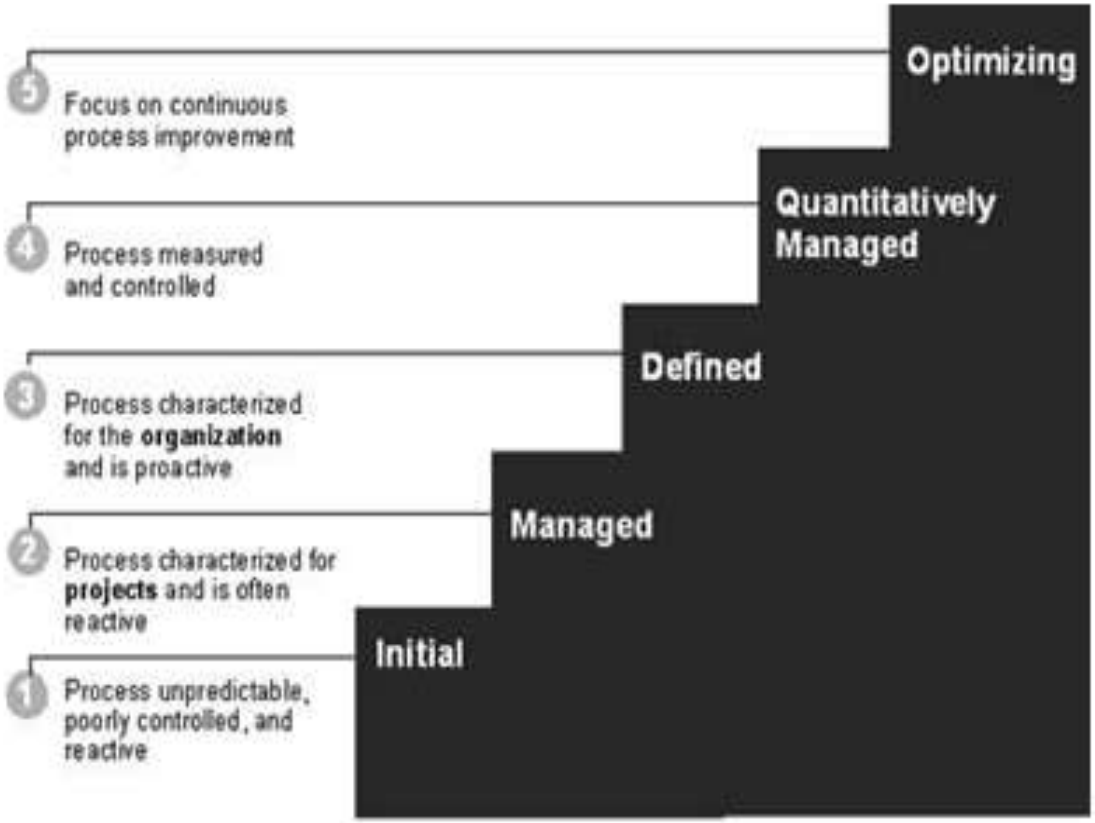
- A maturity model can be viewed as a set of structured levels that describe how well the behaviors, practices and processes of an organization can reliably and sustainably produce required outcomes.
- A maturity model can be used as a benchmark for comparison and as an aid to understanding - for example, for comparative assessment of different organizations where there is something in common that can be used as a basis for comparison. In the case of the CMM, for example, the basis for comparison would be the organizations' software development processes.

Structure

- The model involves five aspects:
- *Maturity Levels:* a 5-level process maturity continuum - where the uppermost (5th) level is a notional ideal state where processes would be systematically managed by a combination of process optimization and continuous process improvement.
- *Key Process Areas:* a Key Process Area identifies a cluster of related activities that, when performed together, achieve a set of goals considered important.
- *Goals:* the goals of a key process area summarize the states that must exist for that key process area to have been implemented in an effective and lasting way. The extent to which the goals have been accomplished is an indicator of how much capability the organization has established at that maturity level. The goals signify the scope, boundaries, and intent of each key process area.
- *Common Features:* common features include practices that implement and institutionalize a key process area. There are five types of common features: commitment to perform, ability to perform, activities performed, measurement and analysis, and verifying implementation.
- *Key Practices:* The key practices describe the elements of infrastructure and practice that contribute most effectively to the implementation and institutionalization of the area.

Levels

- There are five levels defined along the continuum of the model
- *Initial* (chaotic, ad hoc, individual heroics) - the starting point for use of a new or undocumented repeat process.
- *Repeatable* - the process is at least documented sufficiently such that repeating the same steps may be attempted.
- *Defined* - the process is defined/confirmed as a standard business processes.
- *Managed* - the process is quantitatively managed in accordance with agreed-upon metrics.
- *Optimizing* - process management includes deliberate process optimization/improvement.
- Within each of these maturity levels are Key Process Areas which characterise that level, and for each such area there are five factors: goals, commitment, ability, measurement, and verification


5
Focus on continuous process improvement
Optimizing
4
Process measured and controlled
Quantitatively Managed
3
Process characterized for the **organization** and is proactive
Defined
2
Process characterized for **projects** and is often reactive
Managed
1
Process unpredictable, poorly controlled, and reactive
Initial

- **Level 1** - *Initial (Chaotic)*It is characteristic of processes at this level that they are (typically) undocumented and in a state of dynamic change, tending to be driven in an *ad hoc*, uncontrolled and reactive manner by users or events. This provides a chaotic or unstable environment for the processes.
- **Level 2** - *Repeatable*It is characteristic of processes at this level that some processes are repeatable, possibly with consistent results. Process discipline is unlikely to be rigorous, but where it exists it may help to ensure that existing processes are maintained during times of stress.
- **Level 3** - *Defined*It is characteristic of processes at this level that there are sets of defined and documented standard processes established and subject to some degree of improvement over time. These standard processes are in place (i.e., they are the AS-IS processes) and used to establish consistency of process performance across the organization.

- **Level 4** - *Managed*It is characteristic of processes at this level that, using process metrics, management can effectively control the AS-IS process (e.g., for software development). In particular, management can identify ways to adjust and adapt the process to particular projects without measurable losses of quality or deviations from specifications. Process Capability is established from this level.
- **Level 5** - *Optimizing*It is a characteristic of processes at this level that the focus is on continually improving process performance through both incremental and innovative technological changes/improvements.At maturity level 5, processes are concerned with addressing statistical *common causes* of process variation and changing the process (for example, to shift the mean of the process performance) to improve process performance. This would be done at the same time as maintaining the likelihood of achieving the established quantitative process-improvement objectives.

Object-Oriented Metrics

- Object-oriented (OO) metrics are measurements on OO applications used to determine the success or failure of a process or person, and to quantify improvements throughout the software process. These metrics can be used to reinforce good OO programming techniques, which leads to more reliable code.
- the metric is associated with an important external metric, such as reliability, maintainability and fault-proneness [11]. Often these metrics have been used as an early indicator of these externally visible attributes, because the externally visible attributes could not be measures until too late in the software development process.
- Object-Oriented Analysis and Design of software provide many benefits such as reusability, decomposition of problem into easily understood object and the aiding of future modifications. But the OOAD software development life cycle is not easier than the typical procedural approach. Therefore, it is necessary to provide dependable guidelines that one may follow to help ensure good OO programming practices and write reliable code. Object-Oriented programming metrics is an ascpect to be considered. Metrics to be a set of standards against which one can measure the effectiveness of Object-Oriented Analysis techniques in the design of a system

Metrics for OO Software Development Environments

- **Methods per Class**
- Average number of methods per object class = Total number of methods / Total number of object classes
- A larger number of methods per object class complicates testing due to the increased object size and complexity
- if the number of methods per object class gets too large extensibility will be hard
- A large number of methods per object class may be desirable because subclasses tend to inherit a larger number of methods from superclasses and this increases code reuse.

- **Inheritance Denpendencies**
- Inheritance decreases complexity by reducing the number of operations and operators, but this abstraction of objects can make maintenance and design difficult. The two metrics used to measure the amount of inheritance are the depth and breadth of the inheritance hierarchy.
- *2.4.1 Depth of Inheritance Tree (DIT)*
- The depth of a class within the inheritance hierarchy is defined as the maximum length from the class node to the root/parent of the class hierarchy tree and is measured by the number of ancestor classes. In cases involving multiple inheritance, the DIT is the maximum length from the node to the root of the tree [8].

- Well structured OO systems have a forest of classes rather than one large inheritance lattice. The deeper the class is within the hierarchy, the greater the number of methods it is likely to inherit, making it more complex to predict its behavior and, therefore, more fault-prone [15]. Deeper trees require greater design complexity, since more methods and classes are involved [8]. Indeed, deep hierarchies are also a conceptual integrity concern because it becomes difficult to determine which class to specialize from [3]. Additionally, interface changes within the tree must be reflected throughout the entire class tree and object instances. However, the deeper a particular tree is in a class, the greater potential reuse of inherited methods [8].
- Applications can be considered to be "top heavy" if there are too many classes near the root, and indication that designers may not be taking advantage of reuse of methods through inheritance. Alternatively, applications can be considered to be "bottom heavy" whereby too many classes are near the bottom of the hierarchy, resulting in concerns related to design complexity and conceptual integrity.
- *2.4.2 Number of Children (NOC)*
- This metric is the number of direct descendants (subclasses) for each class. Classes with large number of children are considered to be difficult to modify and usually require more testing because of the effects on changes on all the children. They are also considered more complex and fault-prone because a class with numerous children may have to provide services in a larger number of contexts and therefore must be more flexible [3

- Inharitance tree depth = max (inheritance tree path length)
- Inheritance tree depth is likely to be more favorable than breadth in terms of reusability via inheritance. Deeper inheritance trees would seem to promote greater method sharing than would broad trees
- A deep inheritance tree may be more difficult to test than a broad one
- Comprehensibility may be diminished with a large number inheritance layers

Coupling

- The *Coupling Fact*or *(CF)* is evaluated as a fraction. The numerator represents the number of non-inheritance couplings. The denominator is the maximum number of couplings in a system. The maximum number of couplings includes both inheritance and non-inheritance related coupling. Inheritance-based couplings arise as derived classes (subclasses) inherit methods and attributes form its base class (superclass).

- A higher degree of coupling between objects complicates application maintenance because object interconnections and interactions are more complex.
- The higher the degree of uncoupled object the more objects will suitable for reuse within the same applications and within other applications.
- Uncoupled objects should be easier to augment than those with a high degree of 'uses' dependencies, due to the lower degree of interaction.
- Testability is likely to degrade with a more highly coupled system of objects.
- Object interaction complexity associated with coupling can lead to increased error generation during development.

Cohesion

- Cohesion refers to how closely the operations in a class are related to each other.
- Low cohesion is likely to produce a higher degree of errors in the development process. Low cohesion adds complexity which can translate into a reduction in application reliability.
- Objects which are less dependent on other objects for data are likely to be more reusable.

Encapsulation

- Encapsulation means that all that is seen of an object is its interface, namely the operations we can perform on the object [17]."Information hiding is a theoretical technique that indisputably proven its value in practice
- ***Attribute Hiding Factor (AHF)***
- The Attribute Hiding Factor measures the invisibilities of attributes in classes. The invisibility of an attribute is the percentage of the total classes from which the attribute is not visible. An attribute is called visible if it can be accessed by another class or object. Attributes should be "hidden" within a class. They can be kept from being accessed by other objects by being declared a private.
- The Attribute Hiding Factor is a fraction. The numerator is the sum of the invisibilities of all attributes defined in all classes. The denominator is the total number of attributes defined in the project [10]. It is desirable for the Attribute Hiding Factor to have a large value.

- ***Method Hiding Factor (MHF)***
- The Method Hiding Factor measures the invisibilities of methods in classes. The invisibility of a method is the percentage of the total classes from which the method is not visible.
- The Method Hiding Factor is a fraction where the numerator is the sum of the invisibilities of all methods defined in all classes. The denominator is the total number of methods defined in the project.
- Methods should be encapsulated (hidden) within a class and not available for use to other objects. Method hiding increases reusability in other applications and decreases complexity. If there is a need to change the functionality of a particular method, corrective actions will have to be taken in all the objects accessing that method, if the method is not hidden. Thus hiding methods also reduces modifications to the code [10]. The Method Hiding Factor should have a large value.

Complexity

- The larger the number of methods in a class, the greater the potential impact on children since children will inherit all the methods defined in a class. Classes with large numbers of methods are likely to be more application specific, limiting the possibility of reuse. This reasoning indicates that a smaller number of methods is good for usability and reusability.
-

Additional Measures

-
- *2.6.1 Number of Classes*
- If a comparison is made between projects with identical functionality, those projects with more classes are better abstracted.
- *2.6.2 Lines of Code*
- If a comparison is made between projects with identical functionality, those projects with fewer lines of code has superior design and requires less maintenance.
-
- Additionally, methods of large size will always pose a higher risk in attributes such as Understandability, Reusability, and Maintainability.
-

Use Case Estimation

- Before estimating project size, you will first need to configure the technical and environmental factors (see menu item Configuration - Metrics and Estimation Types - TCF and ECF values). For both TCF (technical complexity factor) and ECF (environment complexity factor), an editable table contains a list of factors influencing project productivity. A weight is associated with each factor, reflecting how much that factor relatively affects productivity; a weight is irrelative to a project. The supplied factors and their associated weights are defined by the Use Case Points Method, although they may be adjusted to suit a project's specific needs. For most purposes, the only table column needing adjustment will be 'Value', which indicates the degree of influence a particular factor holds over the project. As a suggested gage, a value of '0' indicates no influence, a '3' indicates average influence, and a '5' indicates strong influence.

- As you build your project using UML use cases to describe the proposed functionality, you should assign a rating to each use case:
- Easy (5 points): The use case is considered a simple piece of work, uses a simple user interface and touches only a single database entity; its success scenario has less than 3 steps; its implementation involves less than 5 classes
- Medium (10 points): The use case is more difficult, involves more interface design and touches 2 or more database entities; its success scenario has between 4 to 7 steps; its implementation involves between 5 to 10 classes
- Complex (15 points): The use case is very difficult, involves a complex user interface or processing and touches 3 or more database entities; its success scenario has over seven steps; its implementation involves more than 10 classes

- he equation is composed of four variables:
- Technical Complexity Factor (TCF).
- Environment Complexity Factor (ECF).
- Unadjusted Use Case Points (UUCP).
- Productivity Factor (PF).
- Each variable is defined and computed separately, using perceived values and various constants. The complete equation is:
- Collapse | Copy Code
- UCP = TCP * ECF * UUCP * PF

Technical Factor

- **Description**
- **Weight**
- **Perceived Complexity**
- **Calculated Factor (weight*perceived complexity)**>
- T1
- Distributed System
- 2
- 5
- 10
- T2
- Performance
- 1
- 4
- 4

- T3
- End User Efficiency
- 1
- 2
- 2
- T4
- Complex internal Processing
- 1
- 4
- 4
- T5

- Reusability
- 1
- 2
- 2
- T6
- Easy to install
- 0.5
- 5
- 2
- T7
- Easy to use
- 0.5
- 3
- 2
- T8
- Portable
- 2
- 3
- 6

- T9
- Easy to change
- 1
- 3
- 3
- T10
- Concurrent
- 1
- 2
- 2
- T11
- Special security features
- 1
- 2
- 2
- T12
- Provides direct access for third parties
- 1
- 5
- 5
- T13
- Special user training facilities are required
- 1
- 3
- 3

Environmental Complexity Factors

- Environmental Complexity estimates the impact on productivity that various environmental factors have on an application. Each environmental factor is evaluated and weighted according to its perceived impact and assigned a value between 0 and 5. A rating of 0 means the environmental factor is irrelevant for this project; 3 is average; 5

- **Environmental Factor**
- **Description**
- **Weight**
- E1
- Familiarity with UML
- 1.5
- E2
- Application Experience
- 0.5
- E3
- Object Oriented Experience
- 1
- E4
- Stable Requirements
- 2
- E7
- Part-time workers
- -1
- E8
- Difficult Programming language
- 2

Unadjusted Use Case Points (UUCP)

- Unadjusted Use Case Points are computed based on two computations:
- The *Unadjusted Use Case Weight* (UUCW) based on the total number of activities (or steps) contained in all the use case Scenarios.
- The *Unadjusted Actor Weight* (UAW) based on the combined complexity of all the use cases Actors.
- **UUCW**

- There is an option to include actors in the estimation calculation; by default, only use cases are considered. If project actors are also included, make sure their complexity has been assigned by some method; rough guidelines to this assignment are supplied below:
- Easy: The actor represents another system with a defined API
- Medium: The actor represents another system interacting through a protocol, like TCP/IP
- Complex: The actor is a person interacting via an interface.