SS ZG622: Software Project Management (Lecture #5)

T V Rao, BITS-Pilani Off-campus Centre, Hyderabad

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

Text Books

T1: Bob Hughes, Mike Cotterel, and Rajib Mall, Software Project Management, 5th Edition, McGraw Hill, 2011

T2: Pressman, R.S. Software Engineering : A Practitioner's Approach, 7th Edition, McGraw Hill, 2010

R1: Sommerville, I., Software Engineering, Pearson Education, 9th Ed., 2010

R2: Capers Jones., Software Engineering Best Practices, TMH ©2010

R3: Robert K. Wysocki, Effective Software Project Management, John Wiley & Sons © 2006

R4: George Stepanek, Software Project Secrets : Why Software Projects Fail, Apress ©2012

R5: A Guide to the Project Management Body of Knowledge (PMBOK® Guide), Fifth Edition by Project Management Institute Project Management Institute © 2013

R6: Introduction to Software Quality by Gerard O'Regan Springer © 2014

innovate achieve lead

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

L5: Sotware Project Management Basics –

Software Metrics– Product, Process, Project, Challenges

Source Courtesy: Some of the contents of this PPT are sourced from materials provided by publishers of prescribed books

Measurement Principles

- The objectives of measurement should be established before data collection begins
- Each technical metric should be defined in an unambiguous manner
- Metrics should be derived based on a theory that is valid for the domain of application (e.g., metrics for design should draw upon basic design concepts and principles and attempt to provide an indication of the presence of an attribute that is deemed desirable)
- Metrics should be tailored to best accommodate specific products and processes

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

A software engineer collects measures and develops metrics in order to obtain indicators

Measurement Process

- *Formulation.* The derivation of software measures and metrics appropriate for the representation of the software that is being considered.
- *Collection.* The mechanism used to accumulate data required to derive the formulated metrics.
- *Analysis.* The computation of metrics and the application of mathematical tools.
- *Interpretation.* The evaluation of metrics results in an effort to gain insight into the quality of the representation.
- *Feedback.* Recommendations derived from the interpretation of product metrics transmitted to the software team.

– Roche, J M "Software Metrics and Measurement Principles" [94]

Metrics Attributes

- *Simple and computable.* It should be relatively easy to learn how to derive the metric, and its computation should not demand inordinate effort or time
- *Empirically and intuitively persuasive.* The metric should satisfy the engineer's intuitive notions about the product attribute under consideration
- *Consistent and objective.* The metric should always yield results that are unambiguous.
- *Consistent in its use of units and dimensions.* The mathematical computation of the metric should use measures that do not lead to bizarre combinations of unit.
- *Programming language independent.* Metrics should be based on the analysis model, the design model, or the structure of the program itself.
- *Effective mechanism for quality feedback.* That is, the metric should provide a software engineer with information that can lead to a higher quality end product

– Ejiogu, L "Software Engineering with Formal Metrics" [91]

Why do we measure ?

- To characterize in order to
  - Gain an understanding of processes, products, resources, and environments
  - Establish baselines for comparisons with future assessments
- To evaluate in order to
  - Determine status with respect to plans
- To predict in order to
  - Gain understanding of relationships among processes and products
  - Build models of these relationships
- To improve in order to
  - Identify roadblocks, root causes, inefficiencies, and other opportunities for improving product quality and process performance

    – Park, Goethert, and Florac [Guidebook on Software Measurement]

- Can be applied to the software process with the intent of improving it on a continuous basis
- Can be used throughout a software project to assist in estimation, quality control, productivity assessment, and project control
- Can be used to help assess the quality of software work products and to assist in tactical decision making as a project proceeds

Etiquette of Process Metrics

- Use common sense and organizational sensitivity when interpreting metrics data
- Provide regular feedback to the individuals and teams who collect measures and metrics
- Don’t use metrics to evaluate individuals
- Work with practitioners and teams to set clear goals and metrics that will be used to achieve them
- Never use metrics to threaten individuals or teams
- Metrics data that indicate a problem should not be considered “negative”
  - Such data are merely an indicator for process improvement
- Don’t obsess on a single metric to the exclusion of other important metrics

– Grady

A Product Metrics Taxonomy

Metrics for the Modeling

Analysis Model

- Functionality delivered
  - Provides an indirect measure of the functionality that is packaged within the software
- Specification quality
  - Provides an indication of the specificity and completeness of a requirements specification

Design Model

- Architectural metrics
  - Provide an indication of the quality of the architectural design
- Component-level metrics
  - Measure the complexity of software components and other characteristics that have a bearing on quality
- Interface design metrics
  - Focus primarily on usability
- Specialized object-oriented design metrics
  - Measure characteristics of classes and their communication and collaboration characteristics

Metrics for Construction

Coding

- Complexity metrics
  - Measure the logical complexity of source code (can also be applied to component-level design)
- Length metrics
  - Provide an indication of the size of the software

Testing

- Statement and branch coverage metrics
  - Lead to the design of test cases that provide program coverage
- Defect-related metrics
  - Focus on defects (i.e., bugs) found, rather than on the tests themselves
- Testing effectiveness metrics
  - Provide a real-time indication of the effectiveness of tests that have been conducted
- In-process metrics
  - Process related metrics that can be determined as testing is conducted

Metrics for the Analysis Model

Function Points

Introduction to Function Points

- First proposed by Albrecht in 1979; hundreds of books and papers have been written on functions points since then
- Can be used effectively as a means for measuring the functionality delivered by a system
- Using historical data, function points can be used to
  - Estimate the cost or effort required to design, code, and test the software
  - Predict the number of errors that will be encountered during testing
  - Forecast the number of components and/or the number of projected source code lines in the implemented system
- Derived using an empirical relationship based on
  1) Countable (direct) measures of the software's information domain
  2) Assessments of the software's complexity

Information Domain Values

- Number of external inputs
  - Each external input originates from a user or is transmitted from another application
  - They provide distinct application-oriented data or control information
  - They are often used to update internal logical files
  - They are not inquiries (those are counted under another category)
- Number of external outputs
  - Each external output is derived within the application and provides information to the user
  - This refers to reports, screens, error messages, etc.
  - Individual data items within a report or screen are not counted separately
- Number of external inquiries
  - An external inquiry is an online input that results in the generation of some immediate software response
  - The response is in the form of an on-line output
- Number of internal logical files
  - Each internal logical file is a logical grouping of data that resides within the application's 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 the application but provides data that may be of use to the application

Function Point Computation

1) Identify/collect the information domain values
2) Complete the table shown below to get the count total
   - Associate a weighting factor (i.e., complexity value) with each count based on criteria established by the software development organization
3) Evaluate and sum up the adjustment factors (see the next two slides)
   - "F_i" refers to 14 value adjustment factors, with each ranging in value from 0 (not important) to 5 (absolutely essential)
4) Compute the number of function points (FP)

$$FP = \text{count total} * [0.65 + 0.01 * \text{sum}(F_i)]$$

Information Domain Value	**Count**		**Weighting Factor**			
			Simple	**Average**	**Complex**	
External Inputs	_____	x	3	4	6	= _____
External Outputs	_____	x	4	5	7	= _____
External Inquiries	_____	x	3	4	6	= _____
Internal Logical Files	_____	x	7	10	15	= _____
External Interface Files	_____	x	5	7	10	= _____
Count total						_____

Value Adjustment Factors

1) Does the system require reliable backup and recovery?
2) Are specialized data communications required to transfer information to or from the application?
3) Are there distributed processing functions?
4) Is performance critical?
5) Will the system run in an existing, heavily utilized operational environment?
6) Does the system require on-line data entry?
7) Does the on-line data entry require the input transaction to be built over multiple screens or operations?

Value Adjustment Factors (continued)

8) Are the internal logical files updated on-line?

9) Are the inputs, outputs, files, or inquiries complex?

10) Is the internal processing complex?

11) Is the code designed to be reusable?

12) Are conversion and installation included in the design?

13) Is the system designed for multiple installations in different organizations?

14) Is the application designed to facilitate change and for ease of use by the user?

Function Point Example

A software system has 3 external inputs (2 simple, 1 complex), 1 external output (average), 1 external inquiry (complex), 1 internal logical file (simple), and 3 external interface files (2 simple, 1 complex). Calculate FP.

Information Domain Value	**Count**	**Weighting Factor**				
		Simple	**Average**	**Complex**		
External Inputs	3	**3(x2)**	4	**6(x1)**	=	12
External Outputs	1	4	**5(x1)**	7	=	5
External Inquiries	1	3	4	**6(x1)**	=	6
Internal Logical Files	1	**7(x1)**	10	15	=	7
External Interface Files	3	**5(x2)**	7	**10(x1)**	=	20
Count total						**50**

$$FP = \text{count total} * [0.65 + 0.01 * \text{sum}(F_i)]$$

Assuming that value adjustment factors sum to 45

$$FP = 50 * [0.65 + (0.01 * 45)]$$

$$FP = 55$$

Interpretation of the FP Number

- Assume that past project data for a software development group indicates that
  - One FP translates into 60 lines of object-oriented source code
  - 12 FPs are produced for each person-month of effort
  - An average of three errors per function point are found during analysis and design reviews
  - An average of four errors per function point are found during unit and integration testing
- This data can help project managers revise their earlier estimates
- This data can also help software engineers estimate the overall implementation size of their code and assess the completeness of their review and testing activities

Metrics for the Design Model

Architectural Design Metrics

- These metrics place emphasis on the architectural structure and effectiveness of modules or components within the architecture
- They are “black box” in that they do not require any knowledge of the inner workings of a particular software component

Hierarchical Architecture Metrics

- Fan out: the number of modules immediately subordinate to the module *i*, that is, the number of modules directly invoked by module *i*
- Structural complexity
  - $S(i) = f^2_{out}(i)$, where f_{out}(i) is the "fan out" of module *i*
- Data complexity
  - $D(i) = v(i)/[f_{out}(i) + 1]$, where v(*i*) is the number of input and output variables that are passed to and from module *i*
- System complexity
  - $C(i) = S(i) + D(i)$
- As each of these complexity values increases, the overall architectural complexity of the system also increases
- This leads to greater likelihood that the integration and testing effort will also increase

– D. N. Card and R. L. Glass

Hierarchical Architecture Metrics (continued)

- Shape complexity
  - `size = n + a`, where `n` is the number of nodes and `a` is the number of arcs
  - Allows different program software architectures to be compared in a straightforward manner
- Connectivity density (i.e., the arc-to-node ratio)
  - `r = a/n`
  - May provide a simple indication of the coupling in the software architecture

– N. Fenton

Metrics for Object-Oriented Design

- Size
  - Population: a static count of all classes and methods
  - Volume: a dynamic count of all instantiated objects at a given time
  - Length: the depth of an inheritance tree
- Coupling
  - The number of collaborations between classes or the number of methods called between objects
- Cohesion
  - The cohesion of a class is the degree to which its set of properties is part of the problem or design domain
- Primitiveness
  - The degree to which a method in a class is atomic (i.e., the method cannot be constructed out of a sequence of other methods provided by the class)

Specific Class-oriented Metrics

- Weighted methods per class
  - The normalized complexity of the methods in a class
  - Indicates the amount of effort to implement and test a class
- Depth of the inheritance tree
  - The maximum length from the derived class (the node) to the base class (the root)
  - Indicates the potential difficulties when attempting to predict the behavior of a class because of the number of inherited methods
- Number of children (i.e., subclasses)
  - As the number of children of a class grows
    - Reuse increases
    - The abstraction represented by the parent class can be diluted by inappropriately defined children
    - The amount of testing required will increase

- CK (Chidamber & Kemerer) Metrics Suite

Specific Class-oriented Metrics

- Coupling between object classes
  - Measures the number of collaborations a class has with any other classes
  - Higher coupling decreases the reusability of a class
  - Higher coupling complicates modifications and testing
  - Coupling should be kept as low as possible
- Response for a class
  - This is the set of methods that can potentially be executed in a class in response to a public method call from outside the class
  - As the response value increases, the effort required for testing also increases as does the overall design complexity of the class
- Lack of cohesion in methods
  - This measures the number of methods that access one or more of the same instance variables (i.e., attributes) of a class
  - If no methods access the same attribute, then the measure is zero
  - As the measure increases, methods become more coupled to one another via attributes, thereby increasing the complexity of the class design

- CK (Chidamber & Kemerer) Metrics Suite

Metrics for Source Code/Testability/Maintenance

Static software product metrics

Software metric	Description
Fan-in/Fan-out	Fan-in is the number of functions or methods that call another function or method (say X). Fan-out is the number of functions that are called by function X. A high fan-in means that X is tightly coupled to the rest of the design. A high fan-out suggests complexity due to control logic.
Length of code	Larger the size of the code of a component, the more complex and error-prone that component is likely to be. Length of code has been a reliable metrics for predicting error-proneness in components.
Cyclomatic complexity	This is a measure of the control complexity of a program. This control complexity may be related to program understandability.
Length of identifiers	Longer the identifiers, the more likely they are to be meaningful and hence the more understandable the program.
Depth of conditional nesting	It is a measure of the depth of nesting of if-statements in a program. Deeply nested if-statements are hard to understand and potentially error-prone.
Fog index	This is a measure of the average length of words and sentences in documents. The higher the value of a document's Fog index, the more difficult the document is to understand.

Halstead Metrics

- *Halstead's Software Science*: a comprehensive collection of metrics, all predicated on the number (count and occurrence) of operators and operands within a component or program.

- It should be noted that Halstead's "laws" have generated substantial controversy, and many believe that the underlying theory has flaws. However, experimental verification for selected programming languages has been performed.

Metrics for Testing

- Testing effort can also be estimated using metrics derived from Halstead measures
- Binder [94] suggests a broad array of design metrics that have a direct influence on the "testability" of an OO system.
  - Lack of cohesion in methods (LCOM).
  - Percent public and protected (PAP).
  - Public access to data members (PAD).
  - Number of root classes (NOR).
  - Fan-in (FIN).
  - Number of children (NOC) and depth of the inheritance tree (DIT).

Metrics for Maintenance

- Software maturity index (SMI)
  - Provides an indication of the stability of a software product based on changes that occur for each release
- $SMI = [M_T - (F_a + F_c + F_d)]/M_T$
  where
  M_T = #modules in the current release
  F_a = #modules in the current release that have been added
  F_c = #modules in the current release that have been changed
  F_d = #modules from the preceding release that were deleted in the current release
- As the SMI (i.e., the fraction) approaches 1.0, the software product begins to stabilize
- The average time to produce a release of a software product can be correlated with the SMI

– IEEE

Metrics in the Process/Project Domain

Process Metrics

The metrics can be

- Quality-related
  - focus on quality of work products and deliverables
- Productivity-related
  - Production of work-products related to effort expended
- Statistical SQA data
  - error categorization & analysis
- Defect removal efficiency
  - propagation of errors from process activity to activity
- Reuse data
  - The number of components produced and their degree of reusability

Metrics in the Process Domain

- Process metrics are collected across all projects and over long periods of time
- They are used for making strategic decisions
- The intent is to provide a set of process indicators that lead to long-term software process improvement
- The only way to know how/where to improve any process is to
  - Measure specific attributes of the process
  - Develop a set of meaningful metrics based on these attributes
  - Use the metrics to provide indicators that will lead to a strategy for improvement

Metrics in the Process Domain(continued)

- We measure the effectiveness of a process by deriving a set of metrics based on outcomes of the process such as
  - Errors uncovered before release of the software
  - Defects delivered to and reported by the end users
  - Work products delivered
  - Human effort expended
  - Calendar time expended
  - Conformance to the schedule
  - Time and effort to complete each generic activity

Metrics in the Project Domain

- Project metrics enable a software project manager to
  - Assess the status of an ongoing project
  - Track potential risks
  - Uncover problem areas before their status becomes critical
  - Adjust work flow or tasks
  - Evaluate the project team's ability to control quality of software work products
- Many of the same metrics are used in both the process and project domain
- Project metrics are used for making tactical decisions
  - They are used to adapt project workflow and technical activities

Use of Project Metrics

- The first application of project metrics occurs during estimation
  - Metrics from past projects are used as a basis for estimating time and effort
- As a project proceeds, the amount of time and effort expended are compared to original estimates
- As technical work commences, other project metrics become important
  - Production rates are measured (represented in terms of models created, review hours, function points, and delivered source lines of code)
  - Error uncovered during each generic framework activity (i.e, communication, planning, modeling, construction, deployment) are measured

Use of Project Metrics (continued)

- Project metrics are used to
  - Minimize the development schedule by making the adjustments necessary to avoid delays and mitigate potential problems and risks
  - Assess product quality on an ongoing basis and, when necessary, to modify the technical approach to improve quality
- Every project should measure:
  - *inputs*—measures of the resources (e.g., people, tools) required to do the work.
  - *outputs*—measures of the deliverables or work products created during the software engineering process.
  - *results*—measures that indicate the effectiveness of the deliverables.

Project Metrics

In summary, project metrics address

- Productivity
  - Minimize development schedule
  - Avoid potential problems and risks
- Quality (improve technical process)
  - As quality improves, defects are minimized
  - As defects go down, the amount of rework required during the project is also reduced
  - As rework goes down, the overall project cost is reduced

Typical Project Metrics

- Effort/time per software engineering task
- Errors uncovered per review hour
- Scheduled vs. actual milestone dates
- Changes (number) and their characteristics
- Distribution of effort on software engineering tasks

Potential Metrics for Agile Projects

Metric	Description	Automation Strategy	Display Considerations	Type	Timeliness
Acceleration	The change in velocity, often calculated over several iterations.	Calculated from velocity.	—	Scalar	Trailing
Age of work items	The amount of time that a work item has existed.	Requires a work item management system.	Typically charted using a histogram or line chart to indicate the number of work items for a given age range. Often categorized by work item type (requirement, defect, and so on)	Trend	Trailing
Blocking work items	A list of work items blocking other work items.	Requires an electronic Kanban-style taskboard.	On a taskboard a blocking work item is shown in a warning color, such as red. As a list of the blocking work items and optionally the items that are being blocked. In diagram form as a directed graph.	List	Trailing
Build health	Indicates the status of your build. Can be captured at the level of a subsystem or system.	Requires your build management tool to log basic events.	For the current status this is shown using a status indicator (e.g., green = success, yellow = tests passed but some code analysis warnings, red = compile or tests failed).	Scalar or Trend	Trailing
Business value delivered	Assesses the value being delivered to business stakeholders.	Typically manual as this requires surveys, interviews, or observation to gather this data.	As a trend this is typically shown as a line graph with a numerical estimate of the business value delivered along the vertical axis and time along the horizontal axis.	Scalar or Trend	Trailing

Disciplined Agile Delivery: A Practitioner's Guide to Agile Software Delivery in the Enterprise by Scott Ambler and Mark Lines

Potential Metrics for Agile Projects

Metric	Description	Automation Strategy	Display Considerations	Type	Timeliness
Effort/cost projection	Indicates the amount of time (or money) invested to date and any expected time (or money) to complete.	Requires information about team size, charge rates, sunk costs, and estimated schedule.	As a trend this is typically shown as a line chart or histogram. Cost may be categorized by capital (e.g., investment in hardware or facilities) versus expense (e.g., labor costs) spending. You may choose to compare original estimated cost/effort versus current estimate or actual cost/effort.	Scalar or Trend	Trailing and Predictive
Iteration burndown	Indicates the total estimated time required to implement the remaining work items in a given iteration.	Requires the estimated number of hours of work initially accepted by the team and the number of task hours remaining each day.	You may choose to indicate the "ideal burndown," a straight line from the top of the stack on the first day to zero on the last day of the iteration.	Trend	Predictive and Trailing
Lifecycle traceability	Indicates how well majority artifacts are traced/linked to one another. For example, can you trace a stakeholder goal to a detailed requirement to the supporting design to the implementation code and to the validating tests (and potentially in reverse)?	Requires major artifacts to be captured electronically and linked to one another.	In summary a percentage may be calculated per traceability category (such as requirements-to-test or design-to-code). In detail a collection of listings, one for each traceability category, showing actual elements being traced to one another.	Scalar	Trailing

Disciplined Agile Delivery: A Practitioner's Guide to Agile Software Delivery in the Enterprise by Scott Ambler and Mark Lines

Potential Metrics for Agile Projects

Metric	Description	Automation Strategy	Display Considerations	Type	Timeliness
Ranged release burndown	Shows the amount of work completed to date by iteration, the amount of work added each iteration, and a range estimate of the amount of time remaining to complete the current work.	Requires initial size of work items chosen for the release, size of work items completed each iteration, and size of work items added or removed each iteration.	.	Trend	Predictive and Trailing
Release burndown	Indicates the total estimated time in terms of number of iterations required to implement the remaining work items for a given release.	Requires the estimated number of points of work initially estimated by the team and the number of points of work items remaining at the end of each iteration.	You may choose to indicate the total points of work items for each completed iteration added to the work item list to show additional scope added to the project over time.	Trend	Predictive and Trailing
Risk mitigation	Indicates the rate at which risks are mitigated.	Requires information about individual risks and when they are considered mitigated.	Typically shown as burndown chart where the total magnitude of the remaining risks at the end of an iteration are indicated. New risks identified in an iteration may be indicated separately from previously known risks. May categorize by severity of the risk.	Trend	Trailing

Disciplined Agile Delivery: A Practitioner's Guide to Agile Software Delivery in the Enterprise by Scott Ambler and Mark Lines

Potential Metrics for Agile Projects

Metric	Description	Automation Strategy	Display Considerations	Type	Timeliness
Stakeholder satisfaction	Indicates the level of satisfaction of your stakeholders with the solution provided.	Typically manual as this requires surveys, interviews, or observation to gather this data.	May be displayed as a bar chart indicating the number or percentage of people who gave a certain rating (e.g., 7 people were very satisfied, 85 people satisfied, and so on). May be categorized by stakeholder issue (e.g., consumability, value delivered, timeliness of delivery, ease of deployment, and so on). May be categorized by stakeholder type.	Scalar or Trend	Trailing
Team morale	Indicates the morale of the team.	Typically manual as this information is gathered via surveys, interviews, or observation.	As a trend typically shown as a line chart.	Scalar or Trend	Trailing
Time invested	The amount of time spent on the project.	Typically manual as this requires data entry to categorize work.	As a trend shown as a histogram or line chart indicating the amount of time spent each iteration. May be categorized by activity (e.g., development, training, planning, demos, supporting other teams, and so on).	Scalar or Trend	Trailing
Velocity	The number of points of work completed in an iteration.	Can be automated if the team is using a work item management tool.	As a trend shown as a line chart or histogram indicating the velocity each iteration.	Scalar or Trend	Trailing

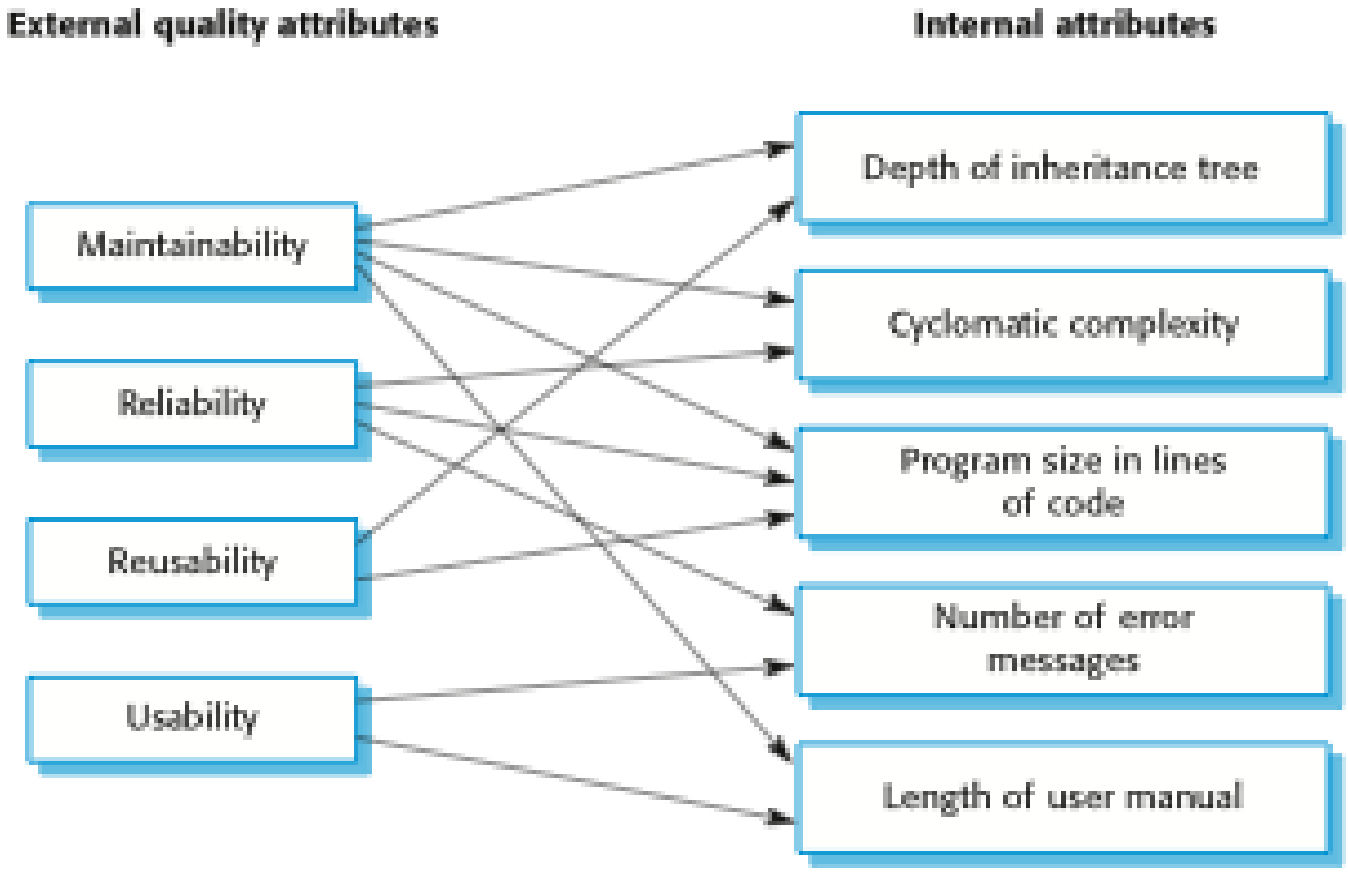
Disciplined Agile Delivery: A Practitioner's Guide to Agile Software Delivery in the Enterprise by Scott Ambler and Mark Lines

Software Measurement Challenges

Metrics assumptions

- A software property can be measured.
- The relationship exists between what we can measure and what we want to know. We can only measure internal attributes but are often more interested in external software attributes.
- This relationship has been formalised and validated.
- It may be difficult to relate what can be measured to desirable external quality attributes.

Relationships between internal and external attributes



Sommerville

Categories of Software Measurement

- Two categories of software measurement
  - Direct measures of the
    - Software process (cost, effort, etc.)
    - Software product (lines of code produced, execution speed, defects reported over time, etc.)
  - Indirect measures of the
    - Software product (functionality, quality, complexity, efficiency, reliability, maintainability, etc.)
- Project metrics can be consolidated to create process metrics for an organization

Size-oriented Metrics

- Derived by normalizing quality and/or productivity measures by considering the size of the software produced
- Thousand lines of code (KLOC) are often chosen as the normalization value
- Metrics include
  - Errors per KLOC - Errors per person-month
  - Defects per KLOC - KLOC per person-month
  - Dollars per KLOC - Dollars per page of documentation
  - Pages of documentation per KLOC
- Size-oriented metrics are not universally accepted as the best way to measure the software process
- Opponents argue that KLOC measurements
  - Are dependent on the programming language
  - Penalize well-designed but short programs
  - Cannot easily accommodate nonprocedural languages
  - Require a level of detail that may be difficult to achieve

Function-oriented Metrics

- Function-oriented metrics use a measure of the functionality delivered by the application as a normalization value
- Most widely used metric of this type is the function point:

$$FP = \text{count total} * [0.65 + 0.01 * \text{sum (value adj. factors)}]$$

- Function point values on past projects can be used to compute, for example, the average number of lines of code per function point (e.g., 60)
- Typical metrics are
  - errors per FP
  - defects per FP
  - $ per FP
  - pages of documentation per FP
  - FP per person-month

Function Point Controversy

- Like the KLOC measure, function point use also has proponents and opponents
- Proponents claim that
  - FP is programming language independent
  - FP is based on data that are more likely to be known in the early stages of a project, making it more attractive as an estimation approach
  - Does not “penalize” inventive (short) implementations that use fewer LOC than other more clumsy versions
- Opponents claim that
  - FP requires some “sleight of hand” because the computation is based on subjective data
  - FP has no direct physical meaning…it’s just a number

Reconciling LOC and FP Metrics

- Relationship between LOC and FP depends upon
  - The programming language that is used to implement the software
  - The quality of the design
- FP and LOC have been found to be relatively accurate predictors of software development effort and cost
  - However, a historical baseline of information must first be established
- LOC and FP can be used to estimate object-oriented software projects
  - However, they do not provide enough granularity for the schedule and effort adjustments required in the iterations of an evolutionary or incremental process
- The table on the next slide provides a rough estimate of the average LOC to one FP in various programming languages

LOC Per Function Point

Language	Average	Median	Low	High
Ada	154	--	104	205
Assembler	337	315	91	694
C	162	109	33	704
C++	66	53	29	178
COBOL	77	77	14	400
Java	55	53	9	214
PL/1	78	67	22	263
Visual Basic	47	42	16	158

www.qsm.com/?q=resources/function-point-languages-table/index.html

Sizing Object-oriented Software

- Number of scenario scripts (i.e., use cases)
  - This number is directly related to the size of an application and to the number of test cases required to test the system
- Number of key classes (the highly independent components)
  - Key classes are defined early in object-oriented analysis and are central to the problem domain
- Number of support classes
  - Support classes are required to implement the system but are not immediately related to the problem domain (e.g., user interface, database, computation)
  - Estimation of the number of support classes can be made from the number of key classes
  - GUI applications have between two and three times more support classes as key classes
  - Non-GUI applications have between one and two times more support classes as key classes
- Number of subsystems
  - A subsystem is an aggregation of classes that support a function that is visible to the end user of a system

– Lorenz & Kidd

Sizing WebApps

- Number of static Web pages (the end-user has no control over the content displayed on the page)
- Number of dynamic Web pages (end-user actions result in customized content displayed on the page)
- Number of internal page links (internal page links are pointers that provide a hyperlink to some other Web page within the WebApp)
- Number of persistent data objects
- Number of external systems interfaced
- Number of static content objects
- Number of dynamic content objects
- Number of executable functions

Harmful vs. Helpful Metrics

Given the fact that bug repairs are the most expensive element in the history of software, these costs should be measured carefully and accurately.

Harmful Metrics	**Helpful Metrics**
cost per defect	function points, (for normalization of data)
lines of code;	Defect removal efficiency
technical debt	

-Capers Jones

Errors and Hazards of improper Metrics

Cost per Defect

- 1. Cost per defect penalizes quality and is always cheapest where the greatest numbers of bugs are found.
- 2. Cost per defect ignores fixed costs. Even with zero defects there will be costs for inspections, testing, static analysis, and maintenance personnel. These costs are either fixed or inelastic and do not change at the same rate as defect volumes.

LOC

- As with the cost per defect metric, the LOC of code metric ignores fixed costs. The mathematical result is that low-level languages such as assembly and C seem to be cheaper and of higher quality than modern high-level languages such as Ruby and MySQL.

-Capers Jones

An Example for Cost per Defect

Type of test	Writing Test Cases	Running Test Cases	Repairing Defects	TOTAL COSTS	Number of Defects	$ per Defect
Unit	**$1,250.00**	**$750.00**	**$18,937.50**	**$20,937.50**	**50**	**$418.75**
Function	**$1,250.00**	**$750.00**	**$7,575.00**	**$9,575.00**	**20**	**$478.75**
Regression	**$1,250.00**	**$750.00**	**$3,787.50**	**$5,787.50**	**10**	**$578.75**
Performance	**$1,250.00**	**$750.00**	**$1,893.75**	**$3,893.75**	**5**	**$778.75**
System	**$1,250.00**	**$750.00**	**$1,136.25**	**$3,136.25**	**3**	**$1,045.42**
Acceptance	**$1,250.00**	**$750.00**	**$378.75**	**$2,378.75**	**1**	**$2,378.75**

Technical Debt – Concept or Metric?

Proposed by Ward Cunningham (developer of the first wiki) in 1992. Widely accepted among agile proponents

With borrowed money you can do something sooner than you might otherwise, but you'll be paying interest.

- Rushing software out the door to get some experience, you would eventually go back and as you learned things about that software you would repay that loan by refactoring the program to reflect your experience as you acquired it.

Borrowing money thinking that you never had to pay it back

- In plenty of cases people would rush software out the door and learn things but never put that learning back into the program.

Payment up-front and in-full

- The traditional waterfall development cycle avoids programming catastrophe by working out a program in detail before programming begins.

Technical Debt

Concept

The mistakes and errors made during development that escape into the real world when the software is released will accumulate downstream costs to rectify.

Hazards

Technical debt suffers from the same problems as cost per defect and lines of code: it ignores fixed costs.

A major problem with technical debt is that it ignores pre-release defect repairs, which are the major cost driver of almost all software applications.

Second, you need to support released software with customer support personnel who can handle questions and bug reports. And you also need to have maintenance programmers standing by to fix bugs when they are reported. This means that even software with zero defects and very happy customers will accumulate post-release maintenance costs that are not accounted for by technical debt.

As per Capers Jones, avoid depending on it as a metric.

Function Point based Cost/Defect Metrics

All costs below are stated per FP. The system is of 100 FP size.

Type of test	Writing Test Cases	Running Test Cases	Repairing Defects	TOTAL $ per FP	Number of Defects
Unit	**$12.50**	**$7.50**	**$189.38**	**$209.38**	**50**
Function	**$12.50**	**$7.50**	**$75.75**	**$95.75**	**20**
Regression	**$12.50**	**$7.50**	**$37.88**	**$57.88**	**10**
Performance	**$12.50**	**$7.50**	**$18.94**	**$38.94**	**5**
System	**$12.50**	**$7.50**	**$11.36**	**$31.36**	**3**
Acceptance	**$12.50**	**$7.50**	**$3.79**	**$23.79**	**1**

-Capers Jones

Top Applications sized using Pattern Matching

	Application	Size in Function Points (IFPUG 4.2)	Language Level	Total Source Code	Lines per Function Point
1	Star Wars missile defense	3,52,330	3.5	3,22,12,992	91
2	Oracle	3,10,346	4	2,48,27,712	80
3	WWMCCS	3,07,328	3.5	2,80,98,560	91
4	U.S. Air Traffic control	3,06,324	1.5	6,53,49,222	213
5	Israeli air defense system	3,00,655	4	2,40,52,367	80
6	SAP	2,96,764	4	2,37,41,088	80
7	NSA Echelon	2,93,388	4.5	2,08,63,147	71
8	North Korean border defenses	2,73,961	3.5	2,50,47,859	91
9	Iran's air defense system	2,60,100	3.5	2,37,80,557	91
10	Aegis destroyer C&C	2,53,088	4	2,02,47,020	80
11	Microsoft VISTA	1,57,658	5	1,00,90,080	64
12	Microsoft XP	1,26,788	5	81,14,400	64
13	IBM MVS	1,04,738	3	1,11,72,000	107
14	Microsoft Office Professional	93,498	5	59,83,891	64
15	Airline reservation system	38,392	2	61,42,689	160

-Capers Jones

Useful Metric - DRE

- Powerful and useful quality metric ever developed is that of "defect removal efficiency" (DRE).
  - DRE metrics were first developed inside IBM in the early 1970's as a method of evaluating the effectiveness of software inspections compared to software testing.
- DRE bring improvements in
  - Software schedules,
  - software development costs,
  - software maintenance costs,
  - customer satisfaction,
  - team morale,
  - stakeholder satisfaction.

Examples of Low and High DRE

	Case A Low Quality		Case B High Quality	
Defect Potential		1,000		1,000
	Efficiency		Efficiency	
Pre-Test Removal				
Static analysis	0.00%	1,000	60.00%	400
Pre-Test inspection	0.00%	1,000	85.00%	60
Test Removal				
Unit test	25.00%	750	30.00%	42
Function test	27.00%	548	33.00%	28
Regression test	25.00%	411	30.00%	20
Performance test	12.00%	361	17.00%	16
Component test	33.00%	242	37.00%	10
System test	35.00%	157	40.00%	6
Acceptance test	15.00%	134	15.00%	5
Delivered defects		134		5
DRE		86.60%		99.50%

-Capers Jones

Integrating Metrics within the Software Process

GQM (Goal-Question-Metric) Paradigm

GQM is a framework for developing a metrics program, proposed by Victor Basili of the University of Maryland, College Park and the Software Engineering Laboratory at the NASA Goddard Space Flight Center

Here three steps for developing metrics are:

- Conceptual level (Goal) : Generate a set of organizational goals
  - What do you want to improve?
- Operational level (Question) : Derive a set of questions relating to the goals
  - Answers provide visibility into meeting the goals
- Quantitative level (Metric) : Develop a set of metrics needed to answer the questions.

GQM (Goal-Question-Metric) Paradigm

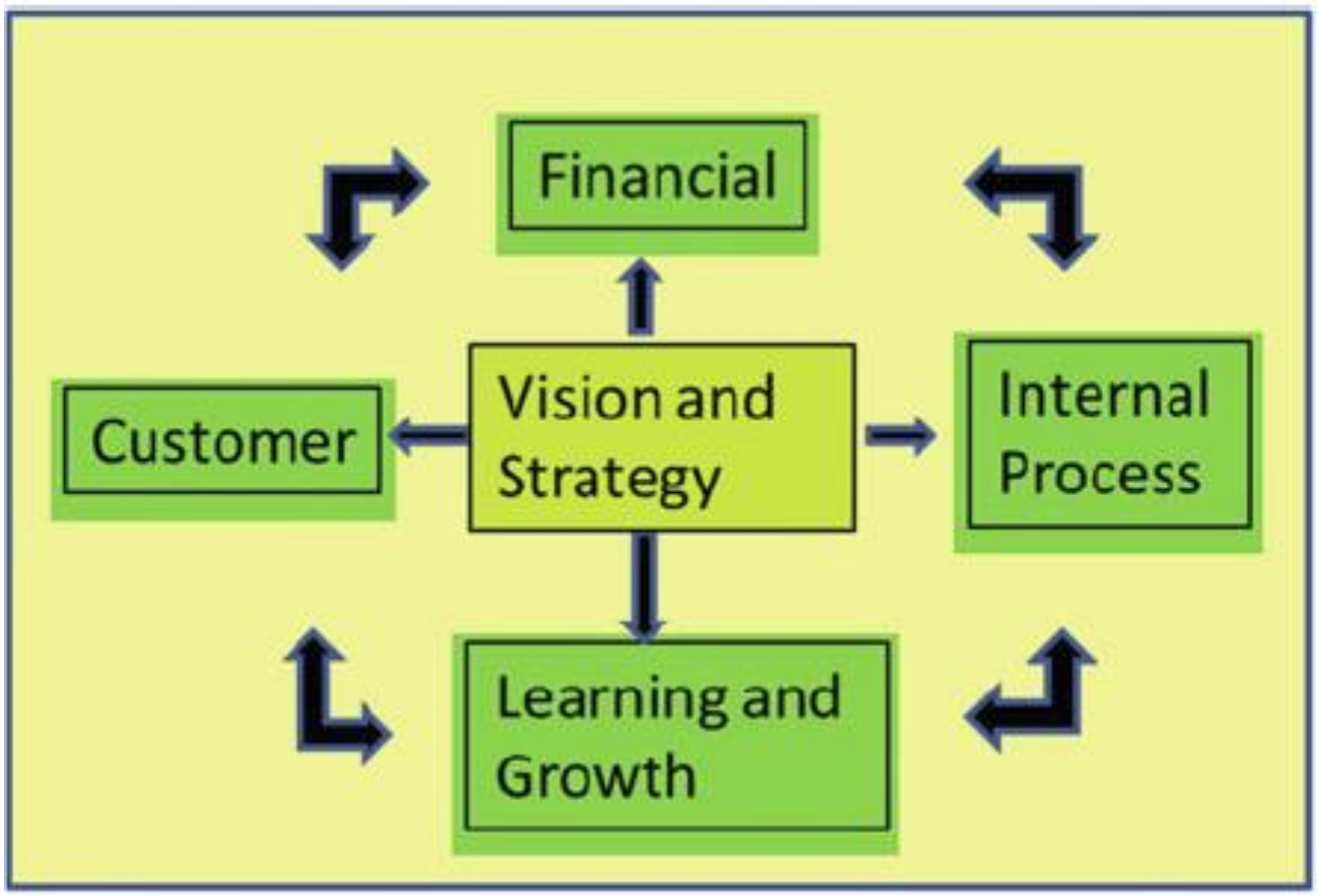
Conceptual level (Goal) : Generate a set of organizational goals

- What do you want to improve?
- Organizational Process Improvement : a) Increase Quality b) Effectively use Inspections

- Operational level (Question) : Derive a set of questions relating to the goals
  - Answers provide visibility into meeting the goals
  - Questions : a) Is defect density being reduced over time? b) What are opportunities for defect prevention? C) How well are inspections performing? D) What is the return of inspections?

- Quantitative level (Metric) : Develop a set of metrics needed to answer the questions.
  - Metrics: defect density trends, defect categorization, inspection efficiency, inspection ROI

The balanced scorecard



The balanced scorecard is a management tool to clarify and translate the organization vision and strategy into action. It was developed by Kaplan and Norton

The balanced scorecard

Financial

Cost of provision of services
Cost of hardware/software
Increase revenue
Reduce costs
Timeliness of solution
99.999 % network availability
24 × 7 customer support

Customer

Quality service
Reliability of solution
Rapid response time
Accurate information
Timeliness of solution
99.999 % network availability
24 × 7 customer support

Internal business process

Requirements elicitation
Software design
Implementation
Testing
Maintenance
Customer support
Security/proprietary information
Disaster prevention and recovery

Learning and growth

Expertise of staff
Software development capability
Project management
Customer support
Staff development career structure
Objectives for staff
Employee satisfaction
Leadership

Challenges with measurement in industry

- It is impossible to quantify the return on investment of introducing an organizational metrics program.
- There are no standards for software metrics or standardized processes for measurement and analysis.
- In many companies, software processes are not standardized and are poorly defined and controlled.
- Most work on software measurement has focused on code-based metrics and plan-driven development processes. However, more and more software is now developed by configuring ERP systems or COTS.
- Introducing measurement adds additional overhead to processes.

-Sommerville

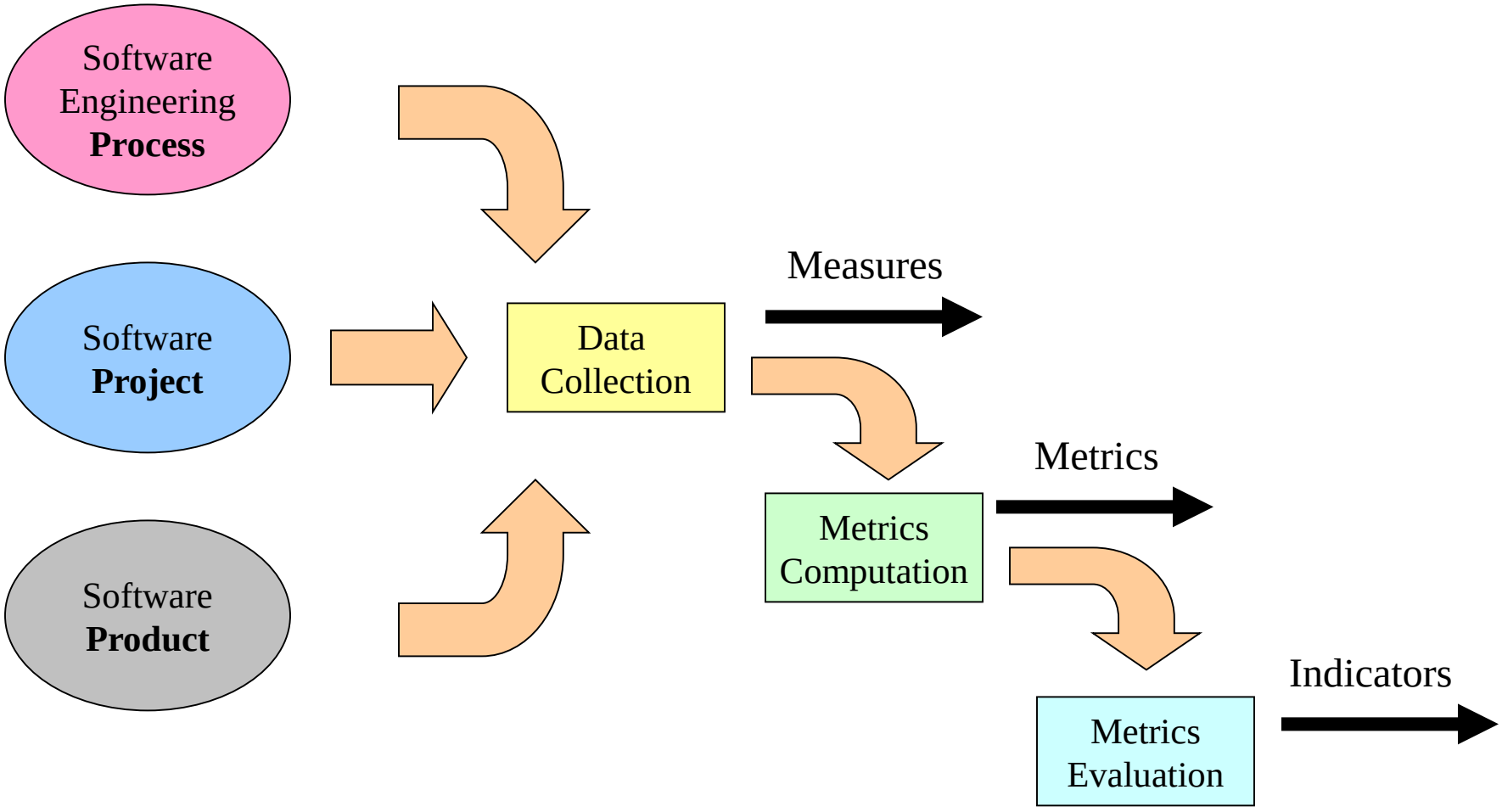
Arguments for Software Metrics

- Most software developers do not measure, and most have little desire to begin
- Establishing a successful company-wide software metrics program can be a multi-year effort
- But if we do not measure, there is no real way of determining whether we are improving
- Measurement is used to establish a process baseline from which improvements can be assessed
- Software metrics help people to develop better project estimates, produce higher-quality systems, and get products out the door on time

Establishing a Metrics Baseline

- By establishing a metrics baseline, benefits can be obtained at the software process, product, and project levels
- The same metrics can serve many masters
- The baseline consists of data collected from past projects
- Baseline data must have the following attributes
  - Data must be reasonably accurate (guesses should be avoided)
  - Data should be collected for as many projects as possible
  - Measures must be consistent (e.g., a line of code must be interpreted consistently across all projects)
  - Past applications should be similar to the work that is to be estimated
- After data is collected and metrics are computed, the metrics should be evaluated and applied during estimation, technical work, project control, and process improvement

Software Metrics Baseline Process



Establishing a Metrics Program

- Identify your business goals.
- Identify what you want to know or learn.
- Identify your subgoals.
- Identify the entities and attributes related to your subgoals.
- Formalize your measurement goals.
- Identify quantifiable questions and the related indicators that you will use to help you achieve your measurement goals.
- Identify the data elements that you will collect to construct the indicators that help answer your questions.
- Define the measures to be used, and make these definitions operational.
- Identify the actions that you will take to implement the measures.
- Prepare a plan for implementing the measures.

(as per SEI Guidebook)

Thank You

Any Questions?