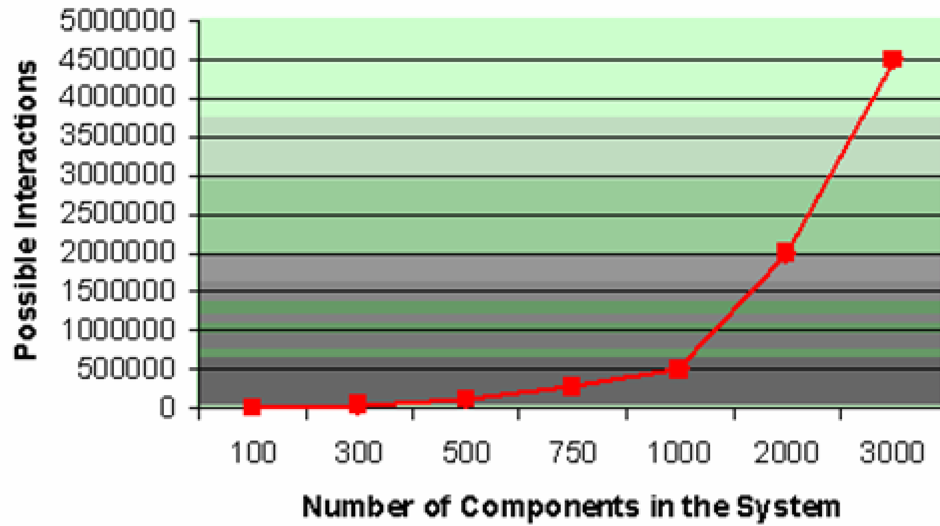
Quality Analysis with Metrics

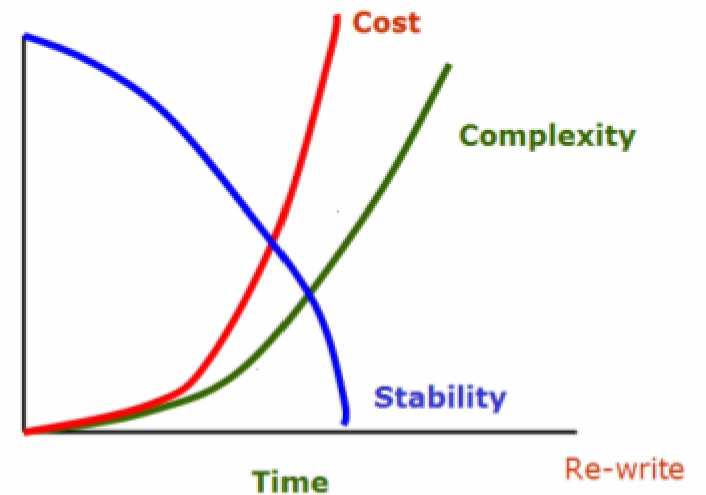
Ameeta Roy
Tech Lead – IBM, India/South Asia

Why do we care about Quality?

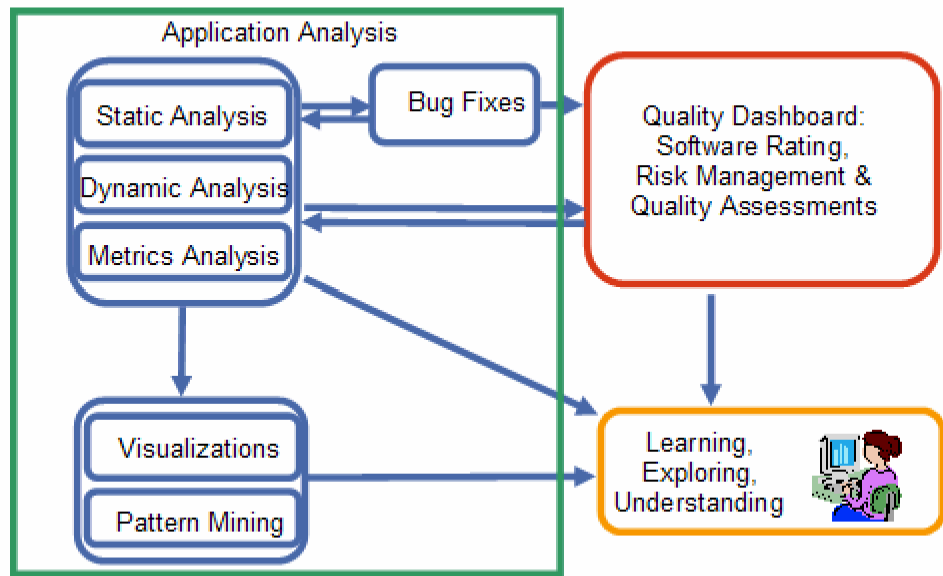
Software may start small and simple, but it quickly becomes complex as more features and requirements are addressed. As more components are added, the potential ways in which they interact grow in a non-linear fashion.



Typical System Life Span



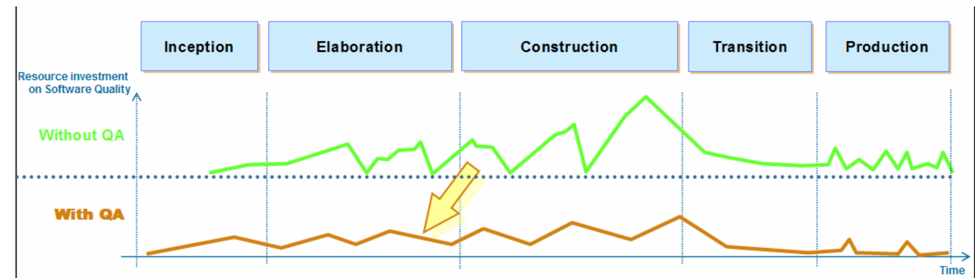
Quality Analysis Stack



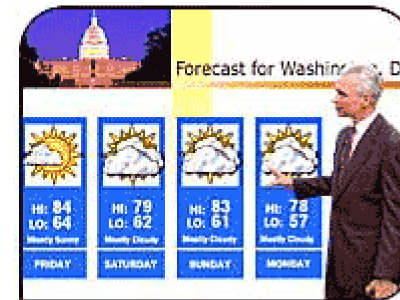
Quality Analysis Phases

- **Assess Quality**
  - Static
    - Architectural Analysis
    - Software Quality Metrics – Rolled UP in to 3 categories
      - Stability
      - Complexity
      - Compliance with Coding Standards
  - Dynamic
    - Performance Criteria
      - Performance,
      - memory consumption

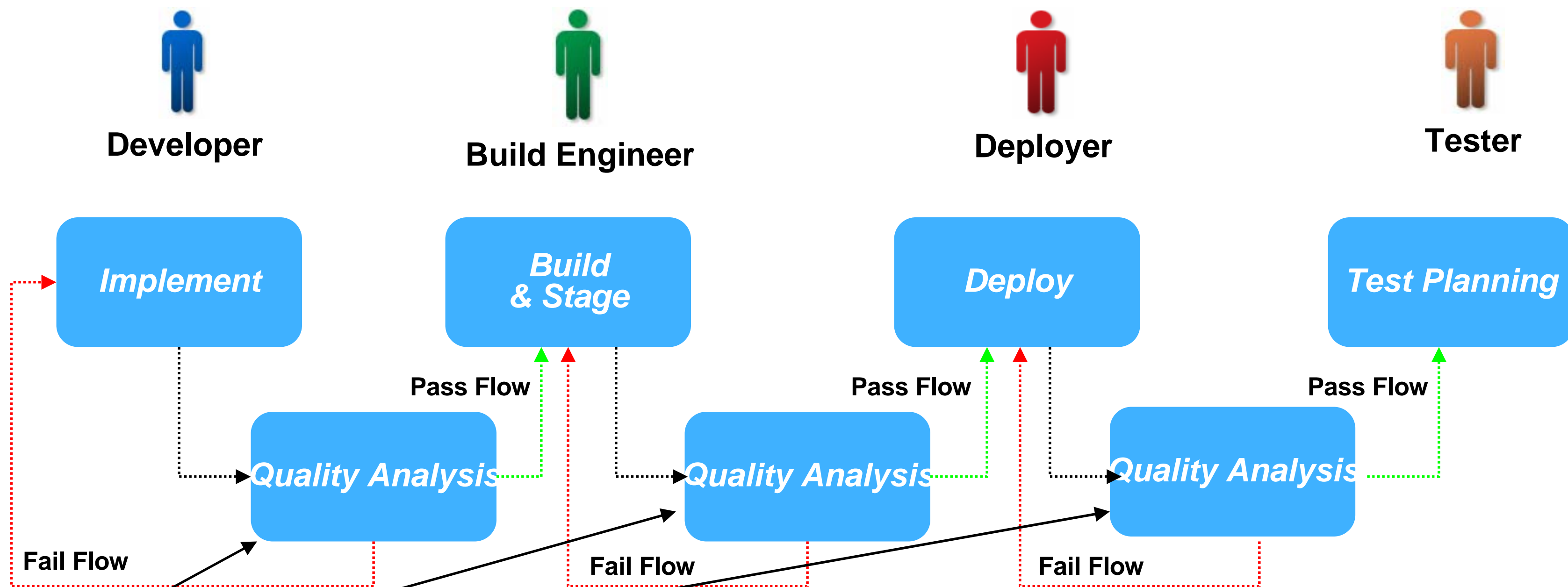
- **Maintain Quality**
  - Static Analysis, Metrics Analysis, Architectural Analysis on every build
  - Testing Efforts
    - Static
      - Statically check test coverage
      - Analyze quality of test cases
      - Prioritize and Compute Testing Activities
    - Dynamic
      - Assess Test Progress
      - Assess Test Effectiveness
      - Dynamically determine code coverage
      - Run Dynamic Analysis with Static Analysis Combination during Testing phase
  - Track the basic project related metrics
    - Churn Metrics (requirements, test cases, code)
    - Defects Metrics(fix rate, introduction rate)
    - Agile metrics for Process
    - Customer Satisfaction (based on surveys, etc.)
    - Costs

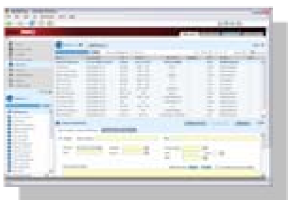


- **Forecast Quality**
  - Number of open defects per priority
  - Defect creation rate
  - Code, requirements churn
  - Defect density compared to project history



Continuous Quality Analysis





QA Lead

1. Configures/Deploys Tool and Rules
2. Defines Pass/Fail Criteria as a function of N metric buckets and thresholds
3. Runs the analysis tool
4. Tool persists the analysis artifacts into DB
5. Tool produces and aggregates metrics for available buckets
6. QA Lead sets up checkpoints, thresholds and pass/fail criteria

Assess Quality via Metrics Analysis

Property	Value
Number of Objects	12
Number of Packages	2
Number of Relationships	52
Maximum Dependencies	14
Minimum Dependencies	0
Average Dependencies	4.33
Maximum Dependents	11
Minimum Dependents	0
Average Dependents	4.33
Relationship To Object Ratio	4.33
Affects on Average	6.8

Java Software Metrics

View as: By Rule

Rule	Metric
Basic Metrics	
Average lines of code per method	0.55
Average number of comments	34.43
Average number of constructors per class	1.96
Average number of methods	10.27
Comment/Code Ratio	9.64 %
Number of attributes	152
Number of comments	792
Number of constructors	200
Number of methods	1048
Cohesion Metrics	
Lack of cohesion 1	3
Lack of cohesion 2	0.73
Lack of cohesion 3	0.71
Complexity Metrics	
Average block depth	1.31
Cyclomatic complexity	1.35
Maintainability index	266.69
Weighted methods per class	1784.00
Dependency Metrics	
Instability	0.65
Normalized Distance	-0.19
Halstead Metrics	
Difficulty level	512.07
Effort to implement	153703606.75
Number of delivered bugs	95.65
Number of operands	21390
Number of operators	9938
Program length	31328
Program level	0.00
Program volume	300159.85
Inheritance Metrics	

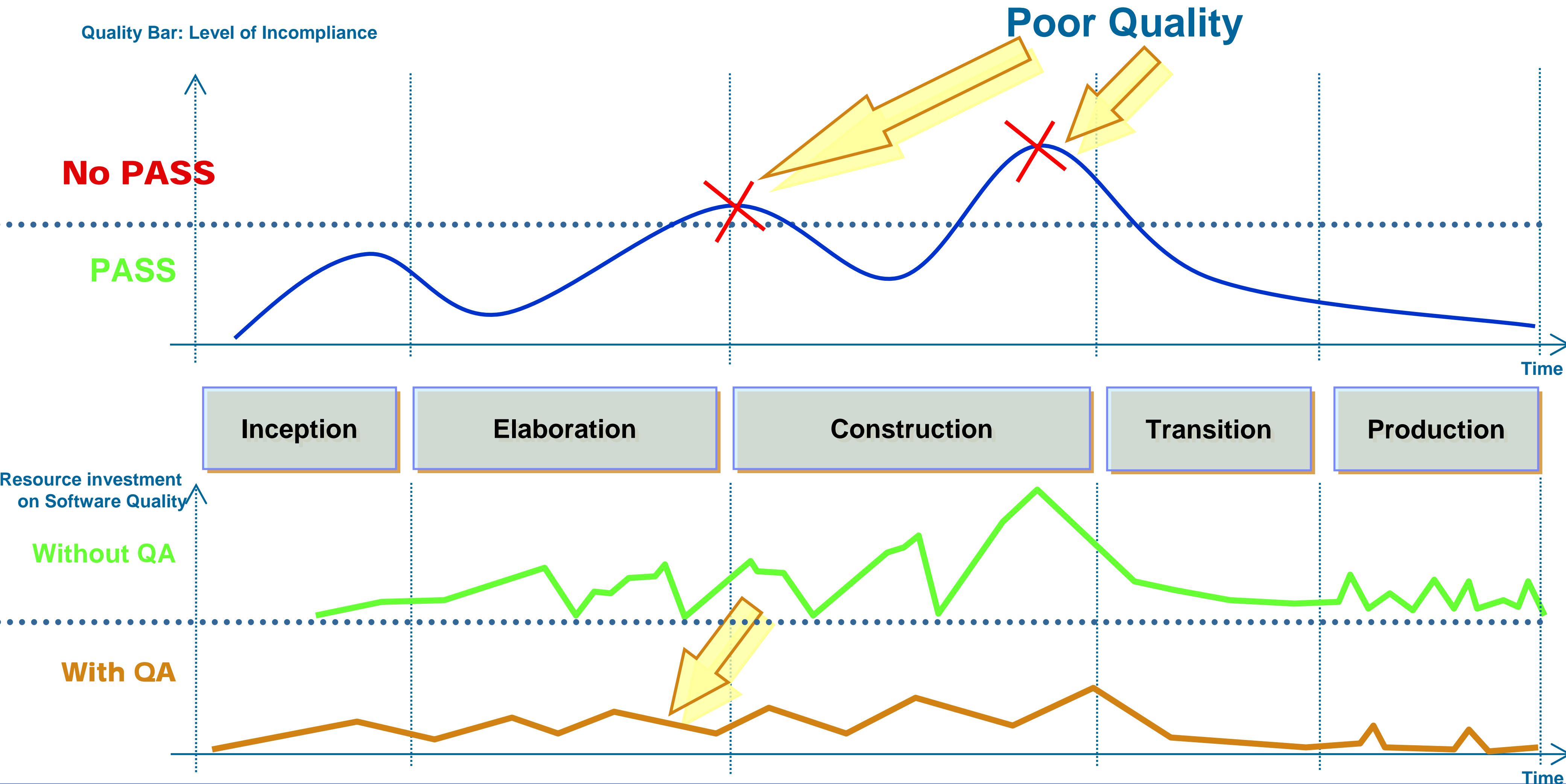
Maintain Quality through Metrics Analysis

Striving for:

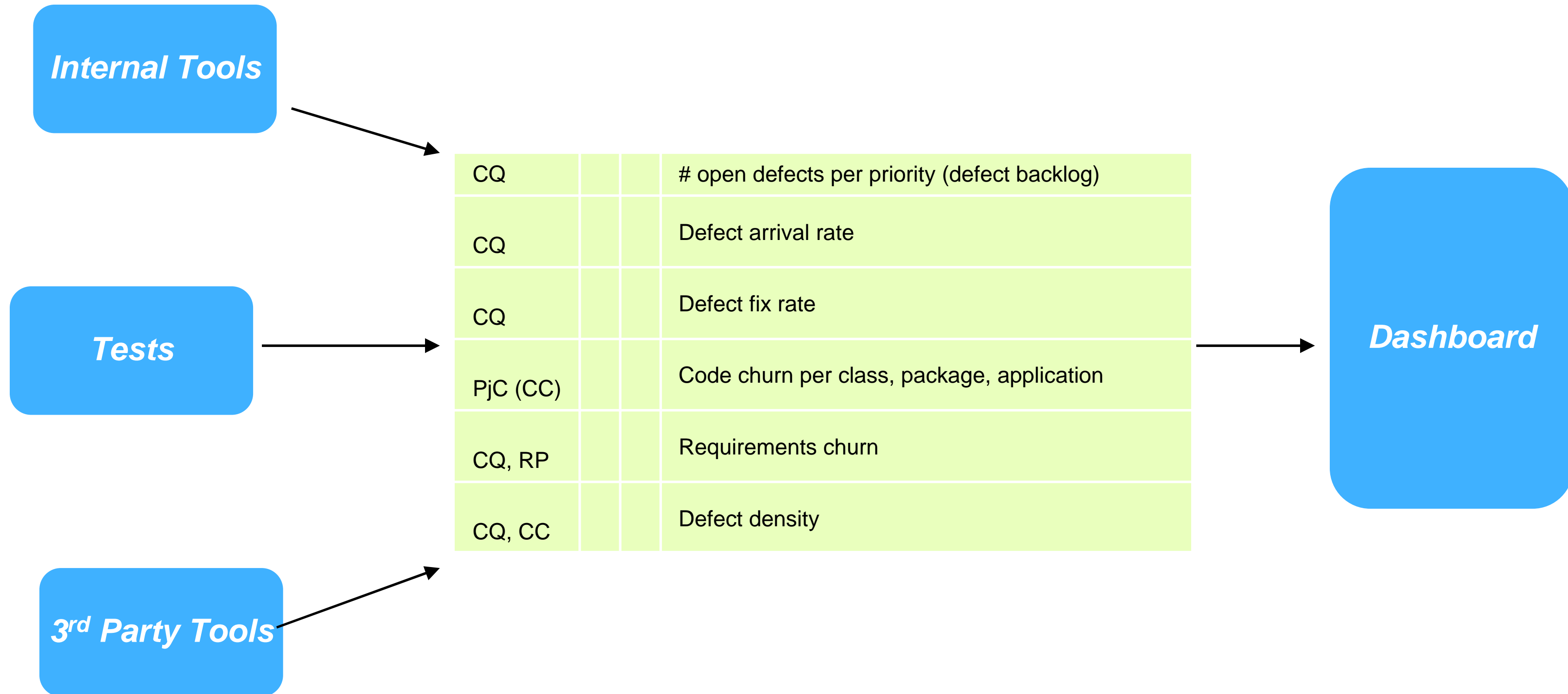
- Above 90% Code Coverage
- Above 90% Complexity Stability
- Above 90% Compliance with Major SE Metrics
- Above 90% Static Analysis Compliance

Recipe for successful release:

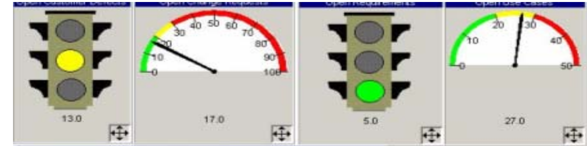
- SA & Unit testing run on every build
- Break flow on checkpoints – do not allow failures
- Continue only when passed



Forecast Quality via Metrics Analysis



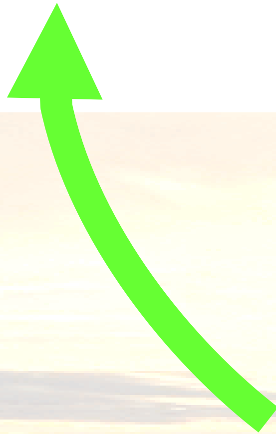
Metrics from Static Analysis

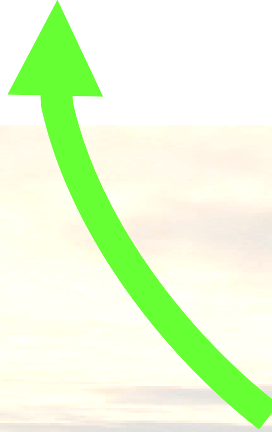


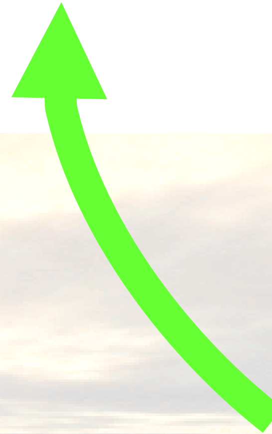
Metric1

Metric2

Metric3







Metrics

Rules

Tests

Assess, Maintain and Forecast Quality through Metrics Roll-up

Project Management Metrics

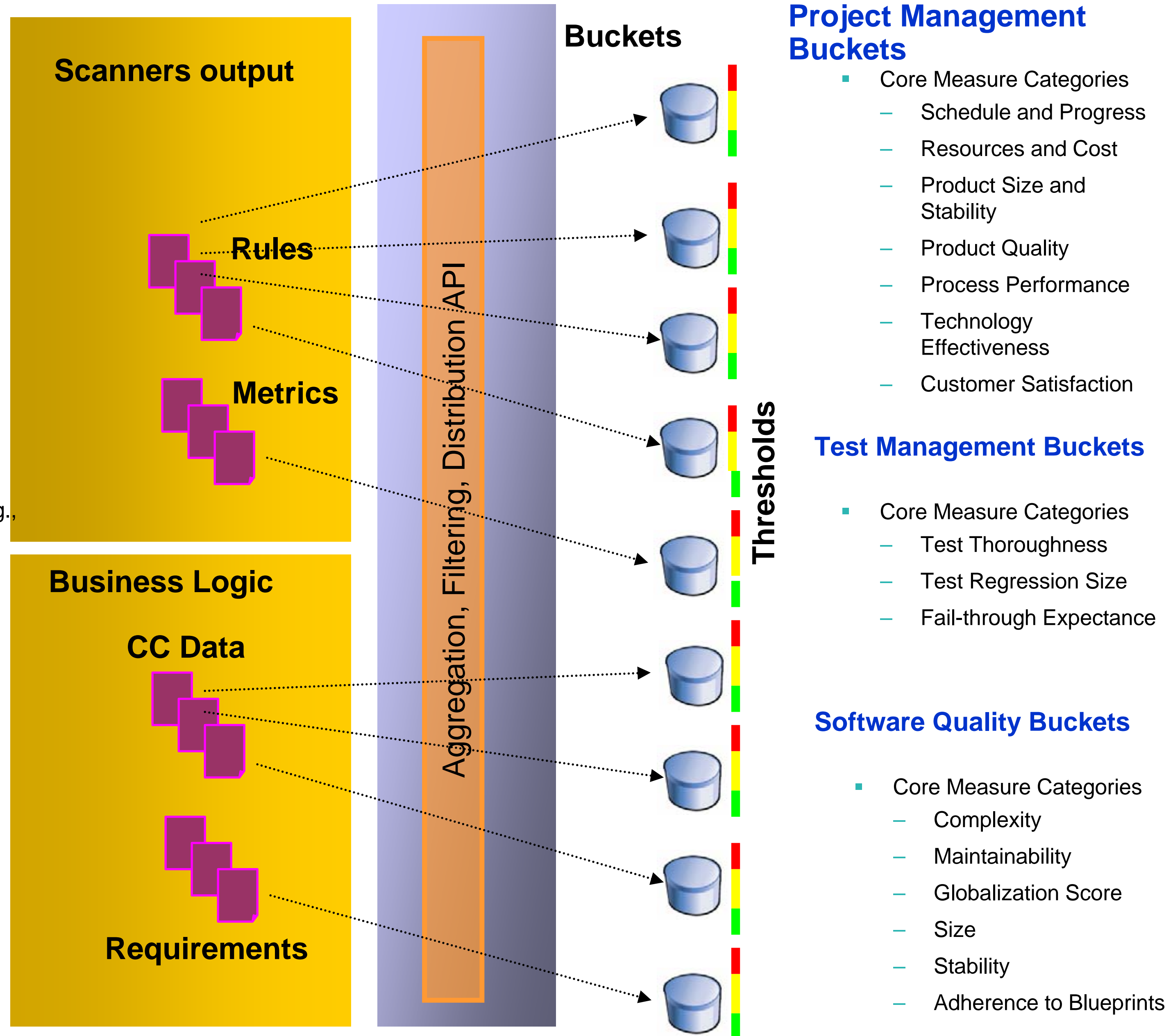
- Forecast quality readiness
  - Number of open defects per priority
  - Defect creation rate
  - Code, requirements churn
  - Defect density compared to project history

Test Management Metrics

- Assess Test Progress
  - Attempted vs. planned tests
  - Executed vs. planned tests
- Assess Test Coverage
  - Code coverage rate (Current, Avg., Min/Max)
  - Object map coverage rate (Current, Avg., Min/Max)
  - Requirements coverage
- Assess Test Effectiveness
  - Test/Case pass/fail rate per execution
  - Coverage per test case
- Prioritize Testing Activities
  - Open defects per priority
  - Planned tests not attempted
  - Planned tests attempted and failed
  - Untested requirements

Software Engineering Metrics

- Complexity
- Rules Output Rollup
- Metrics Rollup



Project Management Buckets

- Core Measure Categories
  - Schedule and Progress
  - Resources and Cost
  - Product Size and Stability
  - Product Quality
  - Process Performance
  - Technology Effectiveness
  - Customer Satisfaction

Test Management Buckets

- Core Measure Categories
  - Test Thoroughness
  - Test Regression Size
  - Fail-through Expectance

Software Quality Buckets

- Core Measure Categories
  - Complexity
  - Maintainability
  - Globalization Score
  - Size
  - Stability
  - Adherence to Blueprints

SE Metrics

	Assess software quality
CQ	# of defects per severity
RAD, RPA, P+	Runtime metrics per method, class, package, application, and test case
RAD, RPA, P+	Execution time (avg. or actual)
RAD, RPA, P+	Memory consumption (avg. or actual)
RSA	SE Metrics
RAD, RSA	# static analysis issues

- Basic Metrics [17/17]
  - Average lines of code per method
  - Average number of attributes per class
  - Average number of comments
  - Average number of constructors per class
  - Average number of methods
  - Average number of parameters
  - Comment/Code Ratio
  - Lines of code
  - Number of attributes
  - Number of comments
  - Number of constructors
  - Number of import statements
  - Number of interfaces
  - Number of lines
  - Number of methods
  - Number of parameters
  - Number of types per package
- Cohesion Metrics [3/3]
  - Lack of cohesion 1
  - Lack of cohesion 2
  - Lack of cohesion 3
- Complexity Metrics [4/4]
  - Average block depth
  - Cyclomatic complexity
  - Maintainability index
  - Weighted methods per class
- Dependency Metrics [5/5]
  - Abstractness
  - Afferent coupling
  - Efferent coupling
  - Instability
  - Normalized Distance
- Halstead Metrics [12/12]
  - Difficulty level
  - Effort to implement
  - Number of delivered bugs
  - Number of operands
  - Number of operators
  - Number of unique operands
  - Number of unique operators
  - Program length
  - Program level
  - Program vocabulary size
  - Program volume
  - Time to implement
- Inheritance Metrics [1/1]
  - Depth of Inheritance

Project Management Metrics

	Forecast quality readiness
CQ	# open defects per priority (defect backlog)
CQ	Defect arrival rate
CQ	Defect fix rate
PjC (CC)	Code churn per class, package, application
CQ, RP	Requirements churn
CQ, CC	Defect density
	Adjust process according to weaknesses (ODC)
CQ (ODC schema)	Defect type trend over time
CQ, CC	Component/subsystem changed over time to fix a defect
CQ, CC	Impact over time
CQ	Defects age over time
	Assess Unit Test Progress
RAD	cumulative # test cases
RAD	Code coverage rate (Current, Avg., Min/Max)
	Agile Metrics (http://w3.webahead.ibm.com/w3ki/display/agileatibm)
Agile Wiki	% of iterations with Feedback Used
Agile Wiki	% of iterations with Reflections

Test Management Metrics

	Assess Test Progress (assume that UnitTests are not scheduled, planned, traced to requirements)
CQ, RFT, RMT, RPT	cumulative # test cases
CQ	# planned, attempted, actual tests
CQ	Cumulative planned, attempted, actual tests in time
CQ	Cumulative planned, attempted, actual tests in points
	Assess Test Coverage
RAD, RPA, P+	Code coverage rate (Current, Avg., Min/Max)
RFT	Object map coverage rate (Current, Avg., Min/Max)
CQ, RP	Requirements coverage (Current, Avg., Min/Max)
	Assess Test Effectiveness
CQ, RFT, RMT, RPT	Hours per Test Case
CQ	Test/Case pass/fail rate per execution
	Coverage per test case
CQ, RAD, RPA, P+	Code coverage
CQ, RFT	Object map coverage
CQ, RP	Requirements coverage
	Prioritize Testing Activities
CQ	Open defects per priority
CQ	# planned tests not attempted
CQ	# planned tests attempted and failed
CQ, RP	# untested requirements

Coupling Metrics

Afferent Couplings	**Afferent Couplings** This is the number of members outside the target elements that depend on members inside the target elements.
Efferent Couplings	**Efferent Couplings** This is the number of members inside the target elements that depend on members outside the target elements.
Instability	**Instability (I)** Description: I = (Ce ÷ (Ca+Ce))
Number of Direct Dependents	**Includes all Compilation depdencies**
Number of Direct Dependencies	**Includes all Compilation depdencies**
Normalized Cumulative Component Dependencies	**Normalized Cumulative Component Dependency(NCCD)** Normalized cumulative component dependency, NCCD, which is the CCD divided by the CCD of a perfectly balanced binary dependency tree with the same number of components. The CCD of a perfectly balanced binary dependency tree of n components is (n+1) * log2(n+1) - n. http://photon.poly.edu/~hbr/cs903-F00/lib_design/notes/large.html
Coupling between object classes	**Coupling between object classes(CBO).** According to the definition of this measure, a class is coupled to another, if methods of one class use methods or attributes of the other, or vice versa. CBO is then defined as the number of other classes to which a class is coupled. Inclusion of inheritance-based coupling is provisional. http://www.iese.fraunhofer.de/Products_Services/more/faq/MORE_Core_Metrics.pdf Multiple accesses to the same class are counted as one access. Only method calls and variable references are counted. Other types of reference, such as use of constants, calls to API declares, handling of events, use of user-defined types, and object instantiations are ignored. If a method call is polymorphic (either because of Overrides or Overloads), all the classes to which the call can go are included in the coupled count. High CBO is undesirable. Excessive coupling between object classes is detrimental to modular design and prevents reuse. The more independent a class is, the easier it is to reuse it in another application. In order to improve modularity and promote encapsulation, inter-object class couples should be kept to a minimum. The larger the number of couples, the higher the sensitivity to changes in other parts of the design, and therefore maintenance is more difficult. A high coupling has been found to indicate fault-proneness. Rigorous testing is thus needed. A useful insight into the 'object-orientedness' of the design can be gained from the system wide distribution of the class fan-out values. For example a system in which a single class has very high fan-out and all other classes have low or zero fan-outs, we really have a structured, not an object oriented, system. http://www.aivosto.com/project/help/pm-oo-ck.html
Data Abstraction coupling	**Data Abstraction Coupling** DAC is defined for classes and interfaces. It counts the number of reference types that are used in the field declarations of the class or interface. The component types of arrays are also counted. Any field with a type that is either a supertype or a subtype of the class is not counted. http://maven.apache.org/reference/metrics.html

Information Complexity Metrics

Depth Of Looping	**Depth Of Looping (DLOOP)** **Depth of looping equals the maximum level of loop nesting in a procedure. Target at a maximum of 2 loops in a procedure.** **http://www.aivosto.com/project/help/pm-complexity.html**
Information Flow	**Information Flow (IFIO)** **Fan-in IFIN = Procedures called + parameters read + global variables read** **Fan-out IFOUT = Procedures that call this procedure + [ByRef] parameters written to + global variables written to** **IFIO = IFIN * IFOUT** **http://www.aivosto.com/project/help/pm-complexity.html**
Information Flow Cohesion	**Information-flow-base cohesion (ICH)** **ICH for a method is defined as the** **number of invocations of other methods of the same class, weighted by** **the number of parameters of the invoked method (cf. coupling measure** **ICP above). The ICH of a class is the sum of the ICH values of its methods.** **http://www.iese.fraunhofer.de/Products_Services/more/faq/MORE_Core_Metrics.pdf**

Class Cohesion

Lack of Cohesion	Lack Of Cohesion (LCOM) A measure for the Cohesiveness of a class. Calculated with the Henderson-Sellers method. If (m (A) is the number of methods accessing an attribute A, calculate the average of m (A) for all attributes, subtract the number of methods m and divide the result by (1-m). A low value indicates a cohesive class and a value close to 1 indicates a lack of cohesion and suggests the class might better be split into a number of (sub) classes. http://metrics.sourceforge.net
Lack of Cohesion1	LCOM1 is the number of pairs of methods in the class using no attribute in common.http://www.iese.fraunhofer.de/Products_Services/more/faq/MORE_Core_Metrics.pdf
Lack of Cohesion2	COM2 is the number of pairs of methods in the class using no attributes in common, minus the number of pairs of methods that do. If this difference is negative, however, LCOM2 is set to zero. http://www.iese.fraunhofer.de/Products_Services/more/faq/MORE_Core_Metrics.pdf
Lack of Cohesion3	LCOM3 Consider an undirected graph G, where the vertices are the methods of a class, and there is an edge between two vertices if the corresponding methods use at least an attribute in common. LCOM3 is then defined as the number of connected components of G. http://www.iese.fraunhofer.de/Products_Services/more/faq/MORE_Core_Metrics.pdf
Lack of Cohesion4	LCOM4 Like LCOM3, where graph G additionally has an edge between vertices representing methods m and n, if m invokes n or vice versa. http://www.iese.fraunhofer.de/Products_Services/more/faq/MORE_Core_Metrics.pdf

Halstead Complexity

The Halstead measures are based on four scalar numbers derived directly from a program's source code:

n1 = the number of distinct operators

n2 = the number of distinct operands

N1 = the total number of operators

N2 = the total number of operands

From these numbers, five measures are derived:

Measure	**Symbol**	**Formula**
Program length	N	N= N1 + N2
Program vocabulary	n	n= n1 + n2
Volume	V	V= N * (LOG2 n)
Difficulty	D	D= (n1/2) * (N2/n2)
Effort	E	E= D * V

Cyclomatic Complexity

The cyclomatic complexity of a software module is calculated from a connected graph of the module
(that shows the topology of control flow within the program):

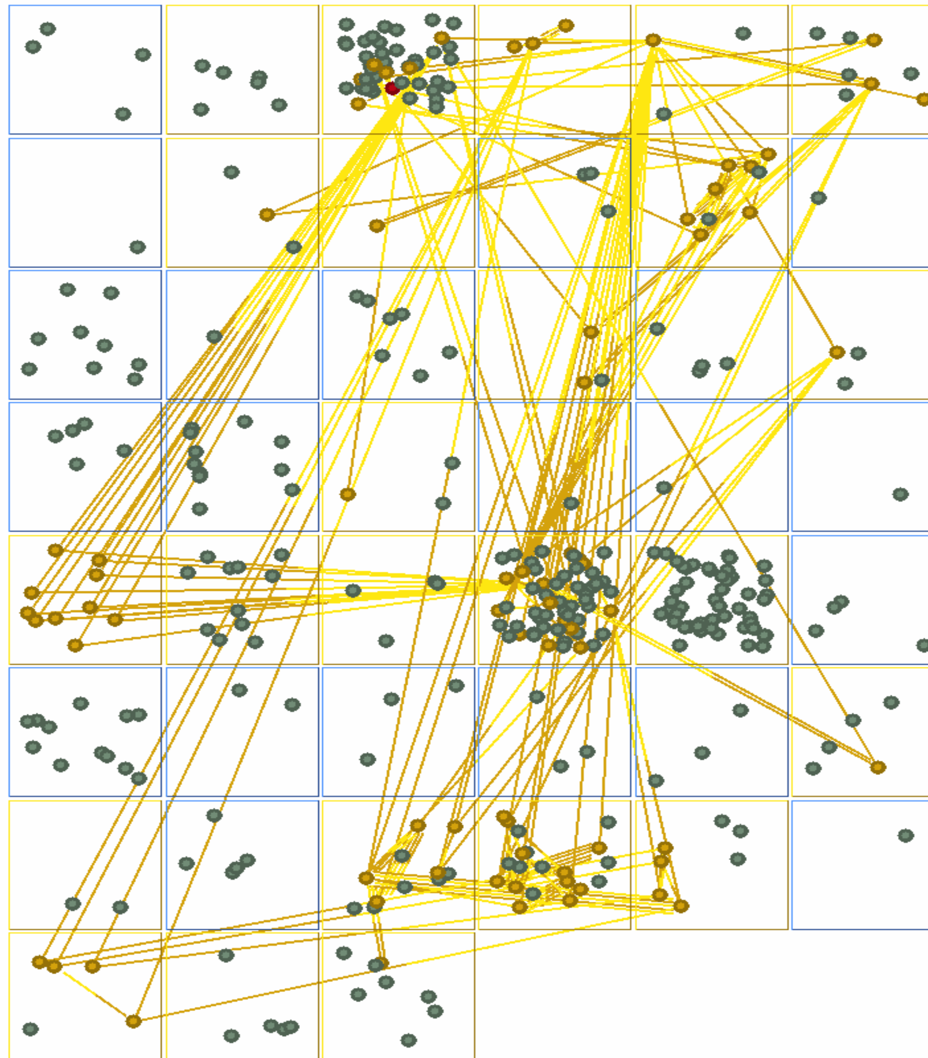
Cyclomatic complexity (CC) = E - N + p
where E = the number of edges of the graph
N = the number of nodes of the graph
p = the number of connected components

Cyclomatic Complexity	Risk Complexity
1-10	a simple program, without much risk
11-20	more complex, moderate risk
21-50	complex, high risk
51+	untestable, very high risk

Cyclomatic Complexity	**Cyclomatic complexity (Vg)** **Cyclomatic complexity is probably the most widely used complexity metric in software engineering. Defined by Thomas McCabe, it's easy to understand, easy to calculate and it gives useful results. It's a measure of the structural complexity of a procedure.** **V(G) is a measure of the control flow complexity of a method or constructor. It counts the number of branches in the body of the method, defined as:** **while statements;** **if statements;** **for statements.** **CC = Number of decisions + 1** **http://www.aivosto.com/project/help/pm-complexity.html** **http://maven.apache.org/reference/metrics.html**
Cyclomatic Complexity2	**Cyclomatic complexity2(Vg2)** **CC2 = CC + Boolean operators** **CC2 includes Boolean operators in the decision count. Whenever a Boolean operator (And, Or, Xor, Eqv, AndAlso, OrElse) is found within a conditional statement, CC2 increases by one.** **The reasoning behind CC2 is that a Boolean operator increases the internal complexity of the branch. You could as well split the conditional statement in several sub-conditions while maintaining the complexity level.** **http://www.aivosto.com/project/help/pm-complexity.html**

SmallWorlds Stability (SA4J)

The stability is calculated as follows. For every component C (class/interface) in the system compute Impact(C) = Number of components that which potentially use C in the computation. That is it is a transitive closure of all relationships. Then calculate Average Impact as Sum of all Impact(C) / Number of components in the system. The stability is computed as an opposite of an average impact in terms of a percentage.



Thank You