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.
Quality Analysis Stack

Application Analysis
- Static Analysis
- Dynamic Analysis
- Metrics Analysis

Bug Fixes

Quality Dashboard:
- Software Rating
- Risk Management
- Quality Assessments

Visualizations
- Pattern Mining

Learning, Exploring, Understanding
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

**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

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Objects</td>
<td>12</td>
</tr>
<tr>
<td>Number of Packages</td>
<td>2</td>
</tr>
<tr>
<td>Number of Relationships</td>
<td>52</td>
</tr>
<tr>
<td>Maximum Dependencies</td>
<td>14</td>
</tr>
<tr>
<td>Minimum Dependencies</td>
<td>0</td>
</tr>
<tr>
<td>Average Dependencies</td>
<td>4.33</td>
</tr>
<tr>
<td>Maximum Dependents</td>
<td>11</td>
</tr>
<tr>
<td>Minimum Dependents</td>
<td>0</td>
</tr>
<tr>
<td>Average Dependents</td>
<td>4.33</td>
</tr>
<tr>
<td>Relationship To Object Ratio</td>
<td>4.33</td>
</tr>
<tr>
<td>Affects on Average</td>
<td>6.8</td>
</tr>
</tbody>
</table>
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

Quality Bar: Level of Incompliance

Without QA

With QA

Resource investment on Software Quality

Poor Quality

PASS

No PASS

Inception | Elaboration | Construction | Transition | Production

Time
Forecast Quality via Metrics Analysis

<table>
<thead>
<tr>
<th></th>
<th>Internal Tools</th>
<th>Tests</th>
<th>3rd Party Tools</th>
<th>CQ</th>
<th># open defects per priority (defect backlog)</th>
<th>CQ</th>
<th>Defect arrival rate</th>
<th>CQ</th>
<th>Defect fix rate</th>
<th>PJC (CC)</th>
<th>Code churn per class, package, application</th>
<th>CQ, RP</th>
<th>Requirements churn</th>
<th>CQ, CC</th>
<th>Defect density</th>
</tr>
</thead>
</table>
Metrics from Static Analysis

Metric1 ➔ Metrics ➔ Rules ➔ Tests

Metric2 ➔

Metric3 ➔
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

---

**Software Quality Buckets**
- Core Measure Categories
  - Complexity
  - Maintainability
  - Globalization Score
- Size
- Stability
- Adherence to Blueprints

---

**Test Management Buckets**
- Core Measure Categories
  - Test Thoroughness
  - Test Regression Size
  - Fail-through Expectance

---

**Project Management Buckets**
- Core Measure Categories
  - Schedule and Progress
  - Resources and Cost
  - Product Size and Stability
  - Product Quality
  - Process Performance
  - Technology Effectiveness
  - Customer Satisfaction
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
## Project Management Metrics

### Forecast quality readiness

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CQ</td>
<td># open defects per priority (defect backlog)</td>
</tr>
<tr>
<td>CQ</td>
<td>Defect arrival rate</td>
</tr>
<tr>
<td>CQ</td>
<td>Defect fix rate</td>
</tr>
<tr>
<td>PJ C (CC)</td>
<td>Code churn per class, package, application</td>
</tr>
<tr>
<td>CQ, RP</td>
<td>Requirements churn</td>
</tr>
<tr>
<td>CQ, CC</td>
<td>Defect density</td>
</tr>
</tbody>
</table>

### Adjust process according to weaknesses (ODC)

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CQ</td>
<td>Defect type trend over time</td>
</tr>
<tr>
<td>(ODC schema)</td>
<td></td>
</tr>
<tr>
<td>CQ, CC</td>
<td>Component/subsystem changed over time to fix a defect</td>
</tr>
<tr>
<td>CQ, CC</td>
<td>Impact over time</td>
</tr>
<tr>
<td>CQ</td>
<td>Defects age over time</td>
</tr>
</tbody>
</table>

### Assess Unit Test Progress

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAD</td>
<td>cumulative # test cases</td>
</tr>
<tr>
<td>RAD</td>
<td>Code coverage rate (Current, Avg., Min/Max)</td>
</tr>
</tbody>
</table>

### Agile Metrics

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agile Wiki</td>
<td>% of iterations with Feedback Used</td>
</tr>
<tr>
<td>Agile Wiki</td>
<td>% of iterations with Reflections</td>
</tr>
</tbody>
</table>

[Agile Metrics](http://w3.webahead.ibm.com/w3ki/display/agileatibm)
**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
- **CQ** 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**
- This is the number of members outside the target elements that depend on members inside the target elements.

**Efferent Couplings**
- This is the number of members inside the target elements that depend on members outside the target elements.

**Instability**
- Instability (I)
  - Description: \( I = \frac{C_e}{C_a + C_e} \)

**Number of Direct Dependents**
- Includes all Compilation dependencies

**Number of Direct Dependencies**
- Includes all Compilation dependencies

**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 \( (n+1) \cdot \log_2(n+1) - n \).
  - [http://photon.poly.edu/~hbr/cs903-F00/lib_design/notes/large.html](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](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](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.
# Information Complexity Metrics

<table>
<thead>
<tr>
<th>Information Complexity Metric</th>
<th>Description</th>
<th>Target/Note</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Depth Of Looping</strong></td>
<td><strong>Depth Of Looping (DLOOP)</strong>: Depth of looping equals the maximum level of loop nesting in a procedure. Target at a maximum of 2 loops in a procedure.</td>
<td></td>
<td><a href="http://www.aivosto.com/project/help/pm-complexity.html">http://www.aivosto.com/project/help/pm-complexity.html</a></td>
</tr>
</tbody>
</table>
| **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](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](http://www.iese.fraunhofer.de/Products_Services/more/faq/MORE_Core_Metrics.pdf) |
## Class Cohesion

<table>
<thead>
<tr>
<th>Lack of Cohesion</th>
<th>Lack Of Cohesion (LCOM)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>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.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Lack of Cohesion1</th>
<th>LCOM1 is the number of pairs of methods in the class using no attribute in common. <a href="http://metrics.sourceforge.net">link</a></th>
</tr>
</thead>
<tbody>
<tr>
<td>Lack of Cohesion2</td>
<td>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. <a href="http://www.iese.fraunhofer.de/Products_Services/more/faq/MORE_Core_Metrics.pdf">link</a></td>
</tr>
<tr>
<td>Lack of Cohesion3</td>
<td>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 ). <a href="http://www.iese.fraunhofer.de/Products_Services/more/faq/MORE_Core_Metrics.pdf">link</a></td>
</tr>
<tr>
<td>Lack of Cohesion4</td>
<td>LCOM4 Like LCOM3, where graph ( G ) additionally has an edge between vertices representing methods ( m ) and ( n ), if ( m ) invokes ( n ) or vice versa. <a href="http://www.iese.fraunhofer.de/Products_Services/more/faq/MORE_Core_Metrics.pdf">link</a></td>
</tr>
</tbody>
</table>
Halstead Complexity

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

- \( n_1 \) = the number of distinct operators
- \( n_2 \) = the number of distinct operands
- \( N_1 \) = the total number of operators
- \( N_2 \) = the total number of operands

From these numbers, five measures are derived:

<table>
<thead>
<tr>
<th>Measure</th>
<th>Symbol</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program length</td>
<td>( N )</td>
<td>( N = N_1 + N_2 )</td>
</tr>
<tr>
<td>Program vocabulary</td>
<td>( n )</td>
<td>( n = n_1 + n_2 )</td>
</tr>
<tr>
<td>Volume</td>
<td>( V )</td>
<td>( V = N \times (\log_2 n) )</td>
</tr>
<tr>
<td>Difficulty</td>
<td>( D )</td>
<td>( D = (n_1/2) \times (N_2/n_2) )</td>
</tr>
<tr>
<td>Effort</td>
<td>( E )</td>
<td>( E = D \times V )</td>
</tr>
</tbody>
</table>


**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

<table>
<thead>
<tr>
<th>Cyclomatic Complexity</th>
<th>Risk Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-10</td>
<td>a simple program, without much risk</td>
</tr>
<tr>
<td>11-20</td>
<td>more complex, moderate risk</td>
</tr>
<tr>
<td>21-50</td>
<td>complex, high risk</td>
</tr>
<tr>
<td>51+</td>
<td>untestable, very high risk</td>
</tr>
</tbody>
</table>

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.

\[ \text{V}(G) = \text{Number of decisions} + 1 \]

For a method or constructor, \(V(G)\) is a measure of the control flow complexity. It counts the number of branches in the body of the method, defined as:

- while statements;
- if statements;
- for statements.

\[ \text{CC} = \text{Number of decisions} + 1 \]

http://www.aivosto.com/project/help/pm-complexity.html
http://maven.apache.org/reference/metrics.html

Cyclomatic complexity2 (Vg2)

Cyclomatic complexity2 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 $\text{Impact}(C) = \text{Number of components that potentially use } C \text{ in the computation}$. That is it is a transitive closure of all relationships. Then calculate Average Impact as $\text{Sum of all Impact}(C) / \text{Number of components in the system}$. The stability is computed as an opposite of an average impact in terms of a percentage.
Thank You