Metrics and Problem Detection

Jorge Ressia

Software is complex.

29% Succeeded
18% Failed
53% Challenged

The Standish Group, 2004

After 50 years, software is not “soft” anymore. It is heavy and difficult to manage.

http://www.standishgroup.com/sample_research/PDFpages/q3-spotlightpdf

For example, if you get a piece of software of 1'000'000 lines of code it would take you 3 months to read it if your reading speed is 2 seconds per line of code.
In most projects, the actual development happens only at the code level, with only little documentation maintenance.

What is the current state?
What should we do?
Where to start?
How to proceed?

actual development
Reverse engineering is analyzing a subject system to:
- identify components and their relationships,
- create more abstract representations.

In most projects, the actual development happens only at the code level, with only little documentation maintenance.

http://dx.doi.org/10.1109/52.43044
This is the background of the talk.
You cannot control what you cannot measure.

Tom de Marco

Metrics are functions that assign numbers to products, processes and resources.

Software metrics are measurements which relate to software systems, processes or related documents.

When you can measure what you are speaking about and express it in numbers, you know something about it; but when you cannot measure, when you cannot express it in numbers, your knowledge is of a meagre and unsatisfactory kind: it may be the beginning of knowledge, but you have scarcely, in your thoughts, advanced to the stage of science.
Metrics compress system traits into numbers.

Let’s see some examples...

Examples of size metrics

- NOM - number of methods
- NOA - number of attributes
- LOC - number of lines of code
- NOS - number of statements
- NOC - number of children

Lorenz, Kidd, 1994
Chidamber, Kemerer, 1994
McCabe cyclomatic complexity (CYCLO) counts the number of independent paths through the code of a function.

McCabe, 1977

✓ It reveals the minimum number of tests to write
✗ Interpretation can't directly lead to improvement action

Weighted Method Count (WMC) sums up the complexity of class' methods (measured by the metric of your choice; usually CYCLO).

Chidamber, Kemerer, 1994

✓ It is configurable, thus adaptable to our precise needs
✗ Interpretation can't directly lead to improvement action

Depth of Inheritance Tree (DIT) is the (maximum) depth level of a class in a class hierarchy.

Chidamber, Kemerer, 1994

✓ Inheritance is measured
✗ Only the potential and not the real impact is quantified
Coupling between objects (CBO) shows the number of classes from which methods or attributes are used.

Chidamber, Kemerer, 1994

- It takes into account real dependencies not just declared ones
- No differentiation of types and/or intensity of coupling

Tight Class Cohesion (TCC) counts the relative number of method-pairs that access attributes of the class in common.

\[ TCC = \frac{2}{10} = 0.2 \]

Bieman, Kang, 1995

- Interpretation can lead to improvement action
- Ratio values allow comparison between systems

Access To Foreign Data (ATFD) counts how many attributes from other classes are accessed directly from a measured class.

Marinescu 2006
Metrics alone do not say anything about the quality of the system.
Problem 1: metrics granularity

capture symptoms, not causes of problems
in isolation,
they don’t lead to improvement solutions

Problem 2: implicit mapping

we don’t reason in terms of metrics,
but in terms of design principles
2 big obstacles in using metrics:

- Thresholds make metrics hard to interpret
- Granularity make metrics hard to use in isolation

Can metrics help me in what I really care for? :)

I want nothing to do with metrics!

Understand the Code
  e.g. "insourced" code you are relocated to a new team

Improve the Code
  e.g. refactor the design to make it portable
  e.g. make my subsystem more flexible to a change of requirements
How to get an initial understanding of a system?

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC</td>
<td>35175</td>
</tr>
<tr>
<td>NOM</td>
<td>3618</td>
</tr>
<tr>
<td>NOC</td>
<td>384</td>
</tr>
<tr>
<td>CYCLO</td>
<td>5579</td>
</tr>
<tr>
<td>NOP</td>
<td>19</td>
</tr>
<tr>
<td>CALLS</td>
<td>15128</td>
</tr>
<tr>
<td>FANOUT</td>
<td>8590</td>
</tr>
<tr>
<td>AHH</td>
<td>0.12</td>
</tr>
<tr>
<td>ANDC</td>
<td>0.31</td>
</tr>
</tbody>
</table>
We need means to compare.

<table>
<thead>
<tr>
<th></th>
<th>ANDC</th>
<th>NOM</th>
<th>20.21</th>
<th>19</th>
<th>0.12</th>
<th>35175</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC</td>
<td>3618</td>
<td>NOM</td>
<td>C</td>
<td>ALLS</td>
<td>15128</td>
<td>F</td>
</tr>
<tr>
<td>CYCLO</td>
<td>9.72</td>
<td>0.56</td>
<td>0.92</td>
<td>5579</td>
<td>9.42</td>
<td>8590</td>
</tr>
</tbody>
</table>

The **Overview Pyramid** provides a metrics overview. 

Lanza, Marinescu 2006
The **Overview Pyramid** provides a metrics overview.  

**Size**

<table>
<thead>
<tr>
<th>CALLS: Number of operation calls</th>
<th>FANOUT: Number of Called Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANDC: Average Number of Derived Classes</td>
<td>AHH: Average Hierarchy Height</td>
</tr>
</tbody>
</table>
The **Overview Pyramid** provides a metrics overview.

Lanza, Marinescu 2006
The **Overview Pyramid** provides a metrics overview.

Lanza, Marinescu 2006

I want nothing to do with metrics!

How do I understand code?

Understand the Code
- e.g. "insourced" code
- you are relocated to a new team

How do I improve code?

Improve the Code
- e.g. refactor the design to make it portable
- e.g. make my subsystem more flexible to a change of requirement

I want to have NOTHING TO DO with metrics! ;-)
Quality is more than 0 bugs.

Breaking design principles, rules and best practices deteriorates the code; it leads to design problems.

Imagine changing just a small design fragment and 33% of all classes would require changes.

Design problems are expensive, frequent, and unavoidable.

How to detect and eliminate them?
God Classes tend to centralize the intelligence of the system, to do everything and to use data from small data-classes.

Riel, 1996
God Classes are complex, are not cohesive, access external data.

Detection Strategies are metric-based queries to detect design flaws.

<table>
<thead>
<tr>
<th>Rule 1</th>
<th>Rule 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Metric 1 &gt; Threshold 1</td>
<td>Metric 2 &lt; Threshold 2</td>
</tr>
</tbody>
</table>

Compose metrics into queries using logical operators

Lanza, Marinescu 2006
A **God Class** centralizes too much intelligence in the system.

- **ATFD**: access to foreign data, counts distinct attributes accessed from other classes
- **LAA**: Locality of attribute accesses
- **FDP**: foreign data providers

An **Envious Method** is more interested in data from a handful of classes.

- **ATFD**: access to foreign data, counts distinct attributes accessed from other classes
- **LAA**: Locality of attribute accesses
- **FDP**: foreign data providers

**Data Classes** are dumb data holders.

- **WOC**: weight of class

**Data Classes** are dumb data holders.

Lanza, Marinescu 2006

- NOAP: number of public attributes
- NOAM: number of accessor methods

**Shotgun Surgery** depicts that a change in an operation triggers many (small) in a lot of different operation and classes.

Lanza, Marinescu 2006

- CM = Changing Methods (Number of calls)
- CC = Changing Classes
What is Code Duplication?

Duplicated Code = Source code segments that are found in different places of a system.
- in different files
- in the same file but in different functions
- in the same function

Detected Problem:
File A contains two copies of a piece of code
File B contains another copy of this code

Possible Solution: Extract Method

All examples are made using Duploc from an industrial case study (1 Mio LOC C++ System)
### Noise Elimination

**Assumption:**
- Code segments are just copied and changed at a few places
- Noise elimination transformation
  - remove white space, comments
  - remove lines that contain uninteresting code elements (e.g., just ‘else’ or ‘}’)

### Enhanced Simple Detection Approach

**Code Comparison Step**
As before, but now
Collect consecutive matching lines into match sequences
Allow holes in the match sequence

**Evaluation of the Approach**

**Advantages**
Identifies more real duplication, language independent

**Disadvantages**
Less simple
Misses copies with (small) changes on every line
Visualization provides insights into the duplication situation
A simple version can be implemented in three days
Scalability issue

Dotplots — Technique from DNA Analysis
Code is put on vertical as well as horizontal axis
A match between two elements is a dot in the matrix
Exact Copies

Copies with variations

Insert / Delete
Rapid Code Elements


**Significant Duplication:**
- It is the largest possible duplication chain uniting all exact clones that are close enough to each other.
- The duplication is large enough.

SEC: Size of Exact Clone measures the size of a clone in terms of lines of code.
SDC: Size of Duplication chain, a duplication chain is a block of duplication composed of exact clones that are close enough to be considered as belonging together.
LB: Line Bias is the distance between two consecutive exact clones.
Follow a clear and repeatable process

Don't reason about quality in terms of numbers!

QA is part of the development process

http://loose.upt.ro/incode

Tudor Gîrba
www.tudorgirba.com

Jorge Ressia

creativecommons.org/licenses/by/3.0/