Research Topics in Software Quality

Evaluating Design using Metrics

“
You cannot control what you cannot measure

Design metrics
- quantify size, complexity, quality to evaluate/predict/analyze/assess
- cyclomatic complexity: #paths through procedure
- coupling of class A: #B[A accesses field/method of B]

Tom de Marco

Acknowledgements

“Pragmatic Design Quality Assessment” by Girba, Lanza, Marinescu
Outline

motivation

basic metrics

OO metrics for coupling and cohesion

using metrics to characterize a system

using metrics to detect design defects
Why measuring software?

“You cannot control what you cannot measure” [Tom de Marco]

Measurement is the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to describe them according to clearly defined rules [Fenton91]

project metrics
quantify dynamics of development process
  e.g., to evaluate/control/predict staff and costs

design metrics
quantify size, complexity, quality of product
  e.g., to evaluate/control/predict maintainability
Basic Metrics

Size

LOC(S) lines of code

language-agnostic, simplest to implement
also used in project metrics

“Measuring programming progress by LOC is like measuring building progress by weight” B. Gates

number of software entities

requires parsing technology

e.g., packages NOP(S)
classes NOC(S)
methods NOM(S)
attributes NAS(S)
Quality Attribute Model

ISO 9126 Quality Attribute Model

[McCall77]
## Influence on Quality Attributes

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Category</th>
<th>Sub-Category</th>
<th>Metric</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>size</td>
<td>LOC</td>
<td></td>
</tr>
<tr>
<td></td>
<td>interface C.</td>
<td>SIZE 2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>structural C.</td>
<td>CC</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>NOM</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>WMC</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>RFC</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Inheritance</td>
<td>DIT</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>NOC</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Coupling</td>
<td>Ca</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>CBO</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>CDBC</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>CDOC</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Ce</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>CF</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>DAC</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>I</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>LD</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>MPC</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>PDAC</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Cohesion</td>
<td>LCOM</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ILCOM</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>SCO</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>LOD</td>
<td></td>
</tr>
</tbody>
</table>

### Main property
- Functionality
- Reliability
- Re-Usability
- Efficiency
- Maintainability
- Portability

### Sub Property
- Suitability
- Accuracy
- Interoperability
- Security
- Fault-tolerance
- Recoveryability
- Reliability
- Understandability for Reuse
- Operability for Reuse
- Attractiveness for Reuse
- Re-Usability Compliance
- Time Behavior
- Resource Utilization
- Efficiency Compliance
- Analyzability
- Changeability
- Stability
- Testability
- Maintainability Compliance
- Adaptability
- Installability
- Co-existence
- Conformance
- Replaceability
- Portability Compliance

### Legend
- not evaluated
- highly directly, directly related
- highly inversely, inversely related
- not related

http://www.arisa.se/compendium/
Basic Metrics

Complexity

**CC(P)** McCabe’s cyclomatic complexity possible paths through procedure

- for structured proc: \( \#\text{edges} - \#\text{nodes} + 2 \) [McCabe76]
- with multiple exits: \( \#\text{decision nodes} - \#\text{exit nodes} + 2 \) [Harrison84]

2 paths (1-2-3, 1-4)

\[
\begin{align*}
&= 4 - 4 + 2 \\
&= 1 - 1 + 2
\end{align*}
\]

high for code with nested conditionals, loops, ...

not necessarily bad, if where expected (intrinsic to problem)

but linearly independent: each includes edges not covered in other
OO Metrics

CK suite

named after seminal work
[Chidamber and Kemerer 94]

**WMC(C)** method count per class
sum of metric(m) for all methods of class
e.g., WMC-unity or WMC-CC

**NOiC(C)** number of immediate children

**DIT(C)** depth in inheritance tree

varies with treatment of library classes and Object
deep classes can reuse more, but behavior difficult to understand
too many immediate children might indicate a missing abstraction
OO Metrics
cohesion and coupling

extent to which functionality in an entity belongs and works together

extent to which separate entities depend upon each other
CBO

CBO metric for coupling

A is coupled to B if A uses a method or field of B

CBO(A) = \{B | A is coupled to B\}

#distinct coupled classes
**OO Metrics**

**LCOM4 metric for cohesion**

**LCOM4**  two methods of same class are related if call each other or access the same field

number of connected components in undirected graph

LCOM: lack of cohesion between methods

LCOM4: refinement by [Hitz,Montazeri95] of LCOM in CK-suite

LCOM4 = 2

LCOM4 = 1  ideal case

[http://www.aivosto.com/project/help/pm-oo-cohesion.html#TCC_LCC]
Common pitfalls when using metrics to characterize a system

not balancing selection of metrics
   NOC alone does not say anything about OO design

misusing metrics
   NOC is no proxy for size
   500 classes could all be large or very small

not correlating metrics
   raw values vs proportions
   50 KLOC / 500 NOC = 2000 LOC per class

missing reference points
   what is “normal” for a specific language
   Smalltalk, C++, Java differ in verbosity and idioms
Overview Pyramid facilitates characterizing system using significant OO metrics

Inheritance

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANDC</td>
<td>0.31</td>
</tr>
<tr>
<td>NOP</td>
<td>19</td>
</tr>
<tr>
<td>NOM</td>
<td>384</td>
</tr>
<tr>
<td>LOC</td>
<td>35175</td>
</tr>
<tr>
<td>CYCLO</td>
<td>5579</td>
</tr>
</tbody>
</table>

Size & Complexity

Coupling

proportion

direct metric

[Lanza, Marinescu 2006]
Overview Pyramid
Size & Complexity metrics and proportions

CYCLO/LOC: conditional complexity to expect => 0.2 = new branch every 5 lines

NOC/NOP: classes per package => fine-grained or coarse-grained packages?

computed proportions are independent of each other can be compared with projects of different size
Overview Pyramid

coupling metrics and proportions

<table>
<thead>
<tr>
<th>CALLS</th>
<th>NOM</th>
<th>FANOUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>3618</td>
<td>418</td>
<td>8590</td>
</tr>
<tr>
<td>15128</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**CALLS/NOM**: how intense is coupling
how many other methods are called on average from each method

**FANOUT/CALLS**: how dispersed is coupling
average number of classes called from method

**CALLS(S)** sum of distinct method invocations for all methods
=> multiple calls in same method counted once

**FANOUT(S)** sum of FANOUT(M) [Lorenz94] for all methods
=> number of classes of which a method is called
Overview Pyramid
inheritance proportions (no direct metrics)

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANDC</td>
<td>0.31</td>
</tr>
<tr>
<td>AHH</td>
<td>0.12</td>
</tr>
</tbody>
</table>

**ANDC(S)** average number of direct subclasses

**AHH(S)** average height of inheritance tree at root classes

=> maximum length from root to deepest subclass
**Overview Pyramid**
reference points for proportions

NOC/NOP: high for Java programs

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANDC</td>
<td>0.31</td>
</tr>
<tr>
<td>AHH</td>
<td>0.12</td>
</tr>
<tr>
<td>NOP</td>
<td>19</td>
</tr>
<tr>
<td>NOM</td>
<td>384</td>
</tr>
<tr>
<td>LOC</td>
<td>35175</td>
</tr>
<tr>
<td>NOM</td>
<td>418</td>
</tr>
<tr>
<td>CALLS</td>
<td>15128</td>
</tr>
<tr>
<td>FANOUT</td>
<td>8590</td>
</tr>
</tbody>
</table>

**thresholds determined**
computing proportions
on diverse corpus of 45 Java and 37 C++ systems
various sizes, domains, open-source and commercial

AVG, typical low (AVG-STDEV), typical high (AVG+STEDV)
Overview Pyramid

e.g., iPlasma
http://loose.upt.ro/reengineering/research/iplasma

e.g., Cultivate

http://sewiki.iai.uni-bonn.de/research/cultivate/
Detecting Design Defects

God Class

centralizes intelligence, does everything, uses data from other data classes

from informal design rules violated by defect

God Class: top-level classes should share work uniformly
beware of classes with much non-communicative behavior
beware of classes that directly access data from other classes

formulate detection rule in terms of metrics and thresholds

access “foreign” data
large and complex classes
low cohesion of methods in class

suspects (many!) still need to be assessed manually, suggest using visualizations

[Lanza, Marinescu 2006]
Detecting Design Defects

relevant examples

empirical studies:
- high occurrence rate
- significant negative impact on quality

cover four major characteristics of good design:
- low coupling
- high cohesion
- moderate complexity
- proper encapsulation

automatically detectable with high accuracy
- good precision
- good recall

<table>
<thead>
<tr>
<th>Design flaw</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>God Class</td>
<td>An excessively complex class that breaks encapsulation by directly using data from other classes.</td>
</tr>
<tr>
<td>Schizophrenic Class</td>
<td>A class that has a very low cohesion, as it defines a large interface that is used by disjoint groups of clients.</td>
</tr>
<tr>
<td>Refused Parent Bequest</td>
<td>A subclass that shows a weak connection to its superclass, by insufficiently using the hierarchy-specific methods and data inherited from the superclass.</td>
</tr>
<tr>
<td>Data Class</td>
<td>A class that does not encapsulate its data and usually does not provide significant functionality, allowing other classes to use its data directly.</td>
</tr>
<tr>
<td>Code Duplication</td>
<td>A method that has a significant number of code lines duplicated with at least another method, from a different class.</td>
</tr>
<tr>
<td>Brain Method</td>
<td>A method that is overly long, with statements showing a deep nesting level and an over-complex branching structure.</td>
</tr>
<tr>
<td>Data Clumps</td>
<td>A long parameter list, which appears over and over again in many other methods throughout the system.</td>
</tr>
<tr>
<td>Intensive Coupling</td>
<td>A method that calls an excessive number of other method from one or several other classes.</td>
</tr>
</tbody>
</table>

[Marinescu 2012]

Marinescu 2012

IBM J. RES. & DEV. VOL. 56 NO. 5 PAPER 9 SEPTEMBER/OCTOBER 2012

Informal description of the design flaws used to instantiate the assessment framework.
Detecting Design Defects

Tool Support
e.g., Cultivate [Jancke10]

implemented using logic program queries

\[
\text{smell}(\text{feature\_envy}, \text{MethodId}) \leftarrow \\
\text{internal\_method}(\text{MethodId}), \\
\text{metric}(\text{method\_access\_to\_foreign\_data}, \text{MethodId}, \text{AFTD}), \\
\text{AFTD} > 2, \%2-5 = \text{few} \\
\text{metric}(\text{method\_locality\_of\_attribute\_access}, \text{MethodId}, \text{LAA}), \\
\text{LAA} < 0.3, \%0.3 = \text{third} \\
\text{metric}(\text{method\_foreign\_data\_providers}, \text{MethodId}, \text{FDP}), \\
\text{FDP} \leq 5, \%2-5 = \text{few}
\]
Detecting Design Defects

Tool Support

e.g., Intooitus InCode