Software Metrics and Problem Detection

Oscar Nierstrasz

Selected material by Mircea Lungu
Roadmap

> **Software Metrics**
  - Size / Complexity Metrics
  - Quality Metrics

> **Metric-Based Problem Detection**
  - Detecting Outliers
  - Encoding Design Problems

> **Moose**
Measurements

A measurement is a *mapping from a domain to a range* using mapping *rules*

A *measure* is a *numerical value* or a symbol assigned during mapping

In Software:
measures = “*metrics*”

Measurement owes its existence to Earth;
Estimation of quantity to Measurement;
Calculation to Estimation of quantity;
Balancing of chances to Calculation;
and Victory to Balancing of chances.
Measurements help you to reason about things using a simplified model rather than the things themselves. (Will these boxes fit in my car?)

https://en.wikipedia.org/wiki/Software_metric

The quotation is from Sun Tzu’s “Art of War.” Sun Tzu claims that measurement is the first step towards victory.

https://en.wikipedia.org/wiki/The_Art_of_War
The Measurement Process

The Goal-Question-Metric model proposes three steps to finding the correct metrics.

1) Establish the **goals** of your maintenance or development project.

2) Derive, for each goal, **questions** that allow you to verify its accomplishment.

3) Find what should be **measured** in order to quantify the answer to the questions.

Targets without clear goals will not achieve their goals clearly.

Gilb’s Principle
The Goal-Question-Metric method was introduced by Victor Basili. It proposes three steps for finding the correct metrics that should be collected during a program.

The GQM model seems too simple to deserve to be called a model. Still, its usefulness becomes clear when realizing that many metrics programs start not with a goal in mind, but with measuring what is easy to measure and end up with bunch of unrelated and non-conclusive measurements.

This situation was addressed also by Kybourg: “If you have no viable theory in which X enters, you have very little motivation to generate a measure of X”.

Roadmap

> Software Metrics
  — Size / Complexity Metrics
  — Quality Metrics

> Metric-Based Problem Detection
  — Detecting Outliers
  — Encoding Design Problems

> Moose
Size Measures

> LOC
> NOM
> NOA
> NOC
> NOP
> ... etc.
The graphic is a *polymetric view* that maps metrics to simple visualizations. This is a “System Complexity View” showing an inheritance hierarchy with NOM (# of methods) mapped to the height of each class node, NOA (# attributes) to the width, and LOC to the colour (black indicates the most LOC).

Many object-oriented metrics were first defined in the early 90s. See, for example: Chidamber, Kemerer, *A Metrics Suite for Object Oriented Design*. IEEE TSE, 1994.

http://dx.doi.org/10.1109/32.295895
http://scgresources.unibe.ch/Literature/SMA/Chid94a-OOMetrics.pdf
Cyclomatic Complexity (CYCLO)

The number of independent linear paths through a program. (McCabe ’77)

+ Measures minimum effort for testing
Cyclomatic Complexity is one of the best-known measures of complexity, which equates complexity with control flow: the more paths through a piece of code, the more complex it is considered to be. This is especially useful for measuring test coverage. (Tests should cover all paths through a program.)

CC can be computed by representing control flow as a directed graph, and computing:

\[ CC = E - N + 2 \]

where \( E \) = # edges and \( N \) = # nodes

Weighted Methods per Class (WMC)

The complexity of a class by summing the complexity of its methods, usually using CYCLO. (Chidamber & Kemerer ’94)

+ A proxy for the time and effort required to maintain a class
Intuition: the larger the complexity of a class, the more difficult its maintenance.
The maximum depth level of a class in a hierarchy. (Chidamber & Kemerer ’94)

+ Inheritance depth is a good proxy for complexity
Access To Foreign Data (ATFD)

ATFD counts how many attributes from other classes are accessed directly from a given class. (Lanza & Marinescu ’06)

+ ATFD summarizes the interaction of a class with its environment.
The visualizations are *class blueprints*. They show five categories of methods and attributes of a single class and their calling/accessing relations. In the first column at the left are *constructors*, followed by *public methods*, *internal methods* (protected or private), *accessors*, and finally *attributes*.

Here we see that the `weightAndPlaceClasses` and `layout` methods of `ClassDiagramLayouter` are very large, and use many accessors and attributes of `ClassDiagramNode`. The latter has little behavior of its own.
Roadmap

> Software Metrics
  — Size / Complexity Metrics
  — Quality Metrics

> Metric-Based Problem Detection
  — Detecting Outliers
  — Encoding Design Problems

> Moose
Coupling Between Object Classes (CBO)

CBO for a class is the number of other classes to which it is coupled. (Chidamber & Kemerer ’94)

+ Meant to assess modular design and reuse
The concept of *coupling* was introduced in a 1974 article by Stevens, Myers, and Constantine. *Structured Design*. IBM Systems Journal, 1974

http://segresources.unibe.ch/Literature/SMA/Stev74a-StructuredDesign.pdf
https://en.wikipedia.org/wiki/Coupling_(computer_programming)

Here we see that the class *Project* has both large *fan-in* (clients using it) and *fan-out* (classes it is a client of), so it is highly coupled to much of the system.
Tight Class Cohesion (TCC)

TCC counts the relative number of method-pairs that access attributes of the class in common. (Bieman & Kang, 95)

+ Can lead to improvement action

\[
\text{TCC} = \frac{4}{10} = 0.4
\]
A class is considered to be cohesive using TCC if its methods access many of the same attributes. TCC counts the pairs of methods that access a common attribute.
Roadmap

> Software Metrics
  — Size / Complexity Metrics
  — Quality Metrics

> Metric-based Problem Detection
  — Detecting Outliers
  — Encoding Design Problems

> Moose
Roadmap

> **Software Metrics**
  ─ Size / Complexity Metrics
  ─ Quality Metrics

> **Metric-based Problem Detection**
  ─ **Detecting Outliers**
  ─ Encoding Design Problems

> Moose
Pattern: Study the Exceptional Entities

**Problem**
— How can you quickly gain insight into complex software?

**Solution**
— *Measure* software entities and *study the anomalous ones*

**Steps**
— Use simple metrics
— Visualize metrics to get an overview
— Browse the code to get insight into the anomalies
This pattern is from the open-source book, “Object-Oriented Reengineering Patterns”. Like design patterns, reengineering patterns encode knowledge mined from experience with practical problems in real software systems. Instead of encoding design experience, however, these patterns express how to reverse engineer and reengineer legacy software systems.

This particular pattern is useful when you want to obtain an initial overview of a complex object-oriented software system. Since the code base may be very large, it is not feasible to read even a small portion of the code. Instead, by applying simple metrics and visualizing the results, your attention may be drawn to anomalous outliers (i.e., exceptional entities) that will tell you interesting things about the system.

http://scg.unibe.ch/download/oorp/
The Overview Pyramid provides a metrics overview.
The Overview Pyramid summarizes several common metrics in a single diagram, indicating whether metric values are outliers or not.
The Overview Pyramid provides a metrics overview.
The Overview Pyramid provides a metrics overview.
CALLS = Number of call sites
FANOUT = Sum of all FANOUTS
Coupling Intensity
The Overview Pyramid provides a metrics overview.

Inheritance

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANDC</td>
<td>0.31</td>
</tr>
<tr>
<td>AHN</td>
<td>0.12</td>
</tr>
</tbody>
</table>
ANDC = Average Number of Derived Classes
AHH = Average Hierarchy Height
The Overview Pyramid provides a metrics overview.
The Overview Pyramid provides a metrics overview.

<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>
<tr>
<td>NOP</td>
<td>19</td>
</tr>
<tr>
<td>NOC</td>
<td>384</td>
</tr>
<tr>
<td>NOM</td>
<td>3618</td>
</tr>
<tr>
<td>NOM</td>
<td>418</td>
</tr>
<tr>
<td>LOC</td>
<td>35175</td>
</tr>
<tr>
<td>CYCLO</td>
<td>5579</td>
</tr>
</tbody>
</table>

- **Outlier Detection**
  - close to high
  - close to average
  - close to low
How to obtain the thresholds?

<table>
<thead>
<tr>
<th></th>
<th>Java</th>
<th></th>
<th></th>
<th>C++</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LOW</td>
<td>AVG</td>
<td>HIGH</td>
<td>LOW</td>
<td>AVG</td>
<td>HIGH</td>
</tr>
<tr>
<td>CYCLO/LOC</td>
<td>0.16</td>
<td>0.20</td>
<td>0.24</td>
<td>0.20</td>
<td>0.25</td>
<td>0.30</td>
</tr>
<tr>
<td>LOC/NOM</td>
<td>7</td>
<td>10</td>
<td>13</td>
<td>5</td>
<td>10</td>
<td>16</td>
</tr>
<tr>
<td>NOM/NOC</td>
<td>4</td>
<td>7</td>
<td>10</td>
<td>4</td>
<td>9</td>
<td>15</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Statistical static analysis of reference systems
Context is important (e.g. programming language)
Lanza and Marinescu advise that you study many software systems and extract statistical rules. The book extracts statistics based on 45 Java systems.
Outliers may be signs of poor quality …
Roadmap

> Software Metrics
  — Size / Complexity Metrics
  — Quality Metrics

> Metric-based Problem Detection
  — Detecting Outliers
  — Encoding Design Problems

> Moose
Design principles come in prose - how to measure them?

Rarely a single metric is sufficient >>> Detection Strategies
Detection Strategies...

> ... are metric based queries for detecting design problems

> (Lanza & Marinescu 2002)
God Classes ...

... tend to **centralize the intelligence** of the system, to **do everything**, and to **use data** from small data-classes
God Classes ...

... tend to **centralize the intelligence** of the system, to **do everything**, and to **use data** from small data-classes.

- **Complexity (WMC)**
- **Lack of cohesion (TCC)**
- **Foreign data usage (ATFD)**
God Classes

Class uses directly more than a few attributes of other classes
ATFD > FEW

Functional complexity of the class is very high
WMC > VERY HIGH

Class cohesion is low
TCC < ONE THIRD

Quantifiers
FEW
MANY
TOP
HIGH
ONE THIRD ...
Data Classes are dumb data holders

Interface of class reveals data rather than offering services
WOC < ONE THIRD

AND

Data Class

Class reveals many attributes and is not complex

---

**WOC - Weight Of a Class**

<table>
<thead>
<tr>
<th>Definition</th>
<th>The number of “functional” public methods divided by the total number of public members (Mar02a)</th>
</tr>
</thead>
</table>
Data Classes are dumb data holders

NOPA = #Public Attributes,
NOAM = #Accessor Methods
Roadmap

> Software Metrics
  — Size / Complexity Metrics
  — Quality Metrics

> Metric-based Problem Detection
  — Detecting Outliers
  — Encoding Design Problems

> Moose *(separate presentation)*
What you should know!

> What is a metric?
> Why goals are important for choosing metrics?
> Why is the context of the metrics important?
> What is the difference between coupling and cohesion?
> What metrics are present on the overview pyramid?
> What do detection strategies detect?
> Why reading code is bad?
> How does Moose solve the data analysis problems?
Can you answer these questions?

> What metrics are important for measuring some of your projects (bachelor’s thesis project, etc…)?
> What are acceptable ranges of CYCLO for different languages?
> Why is it hard to calculate CYCLO for Smalltalk methods?
> What insights can a class blueprint provide besides ATFD?
> What are the strategies of solving code smells detected by detection strategies?
Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)

You are free to:

**Share** — copy and redistribute the material in any medium or format

**Adapt** — remix, transform, and build upon the material for any purpose, even commercially.

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:

**Attribution** — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

**ShareAlike** — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

http://creativecommons.org/licenses/by-sa/4.0/