Software metrics (3)

Alexander Serebrenik
From imperative to OO

- All metrics so far were designed for imperative languages
- Applicable for OO
  - On the method level
  - Also
    - Number of files $\rightarrow$ number of classes/packages
    - Fan-in $\rightarrow$ afferent coupling ($C_a$)
    - Fan-out $\rightarrow$ efferent coupling ($C_e$)
- But do not reflect OO-specific complexity
  - Inheritance, class fields, abstractness, ...

- Popular metric sets
  - Chidamber and Kemerer, Li and Henry, Lorenz and Kidd, Abreu, Martin
• WMC – weighted methods per class
  • Sum of metrics(m) for all methods m in class C
• DIT – depth of inheritance tree
  • java.lang.Object? Libraries?
• NOC – number of children
  • Direct descendents
• CBO – coupling between object classes
  • A is coupled to B if A uses methods/fields of B
  • \( \text{CBO}(A) = | \{ B | A \text{ is coupled to } B \} | \)
• RFC - #methods that can be executed in response to a message being received by an object of that class.
Chidamber and Kemerer

- WMC – weighted methods per class
  - Sum of metrics(m) for all methods m in class C
  - Popular metrics: McCabe’s complexity and unity
  - WMC/unity = number of methods
  - Statistically significant correlation with the number of defects

- WMC/unity
- Dark: Basili et al.
- Light: Gyimothy et al. [Mozilla 1.6]
- Red: High-quality NASA system
Chidamber and Kemerer

- **WMC** – weighted methods per class
  - Sum of metrics(m) for all methods m in class C
  - Popular metrics: McCabe’s complexity and unity
  - WMC/unity = number of methods
  - Statistically significant correlation with the number of defects

- **WMC/unity**
- **Gyimothy et al.**
- **Average**
Depth of inheritance - DIT

• Variants: Were to start and what classes to include?
  • 1, JFrame is a library class, excluded
  • 2, JFrame is a library class, included
  • 7
DIT – what is good and what is bad?

- Three NASA systems
- What can you say about the use of inheritance in systems A, B and C?
- Observation: quality assessment depends not just on one class but on the entire distribution
Average DIT in Mozilla

- How can you explain the decreasing trend?
Other CK metrics

- **NOC** – number of children
- **CBO** – coupling between object classes
- **RFC** - #methods that can be executed in response to a message being received by an object of that class.
- More or less “exponentially” distributed

<table>
<thead>
<tr>
<th>Metric</th>
<th>Our results</th>
<th>[1]</th>
<th>[22]</th>
<th>[21]</th>
</tr>
</thead>
<tbody>
<tr>
<td>WMC</td>
<td>++</td>
<td>+</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>DIT</td>
<td>+</td>
<td>++</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>RFC</td>
<td>++</td>
<td>++</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>NOC</td>
<td>0</td>
<td>++</td>
<td>--</td>
<td>+</td>
</tr>
<tr>
<td>CBO</td>
<td>++</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

Significance of CK metrics to predict the number of faults
Modularity metrics: LCOM

- **LCOM** – lack of cohesion of methods

- **Chidamber Kemerer:**

  \[
  LCOM(C) = \begin{cases} 
  P - Q & \text{if } P > Q \\ 
  0 & \text{otherwise}
  \end{cases}
  \]

  where

  - \( P = \# \text{pairs of distinct methods in } C \) that do not share instance variables
  - \( Q = \# \text{pairs of distinct methods in } C \) that share instance variables

Discriminative ability is insufficient

What about methods that use get/set instead of direct access?
First solution: LCOMN

- Defined similarly to LCOM but allows negative values

\[ LCOMN(C) = P - Q \]
• Method * method tables
  • Light blue: Q, dark blue: P
• Calculate the LCOMs
• Does this correspond to your intuition?
• $m$ – number of methods
• $v$ – number of variables (attrs)
• $m(V_i)$ - #methods that access $V_i$

\[
LCOM = \frac{1 - \sum_{i=1}^{v} m(V_i)}{v - m}
\]

• Cohesion is maximal: all methods access all variables
  \[ m(V_i) = m \] and \[ LCOM = 0 \]

• No cohesion: every method accesses a unique variable
  \[ m(V_i) = 1 \] and \[ LCOM = 1 \]

• Can LCOM exceed 1?
If some variables are not accessed at all, then

\[ m(V_i) = 0 \]

and if no variables are accessed

\[
\left( \frac{1}{v} \sum_{i=1}^{v} m(V_i) \right) - m
\]

\[
= \frac{-m}{1 - m}
\]

\[
= 1 + \frac{1}{m - 1}
\]

Hence

LCOM is undefined for \( m = 1 \)

LCOM \( \leq 2 \)
**Project 6 (commercial human resource system) suggests stabilization, but no similar conclusion can be made for other projects**

*Sato, Goldman, Kon 2007*
### Shortcomings of LCOM [Henderson-Sellers ]

Due to [Fernández, Peña 2006]

<table>
<thead>
<tr>
<th>Variables</th>
<th>$m$</th>
<th>$m$</th>
<th>$m$</th>
<th>$m$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Method-variable diagrams: dark spot = access
- $\text{LCOM}(A) \ ? \ \text{LCOM}(B) \ ? \ \text{LCOM}(C) \ ?$

\[
\left( \frac{1}{v} \sum_{i=1}^{v} m(V_i) \right) - m
\]

\[
\frac{1 - m}{1 - m}
\]
Shortcomings of LCOM [Henderson-Sellers]

Due to [Fernández, Peña 2006]

- All LCOM values are the same: 0.67
  - \( m=4, m(V_i) = 2 \) for all \( i \)

- \( A \) seems to be less cohesive than \( B \) and \( C \)!

\[
\begin{align*}
\left( \frac{1}{v} \sum_{i=1}^{v} m(V_i) \right) - m & \\
& = \frac{1 - m}{1 - m}
\end{align*}
\]
Alternative [Hitz, Montazeri 1995]

• LCOM as the number of strongly connected components in the following graph
  • Vertices: methods
    – except for getters/setters
  • Edge between \( a \) and \( b \), if
    – \( a \) and \( b \) access the same variable

• LCOM values
  • 0, no methods
  • 1, cohesive component
  • 2 or more, lack of cohesion
Alternative [Hitz, Montazeri 1995]

- LCOM as the number of strongly connected components in the following graph:
  - Vertices: methods
    - except for getters/setters
  - Edge between a and b, if
    - a and b access the same variable

- LCOM values:
  - 0, no methods
  - 1, cohesive component
  - 2 or more, lack of cohesion
Experimental evaluation of LCOM variants

<table>
<thead>
<tr>
<th></th>
<th>Correlation with expert assessment</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Group 1</td>
</tr>
<tr>
<td>Cox, Etzkorn and Hughes 2006</td>
<td>-0.43 (p = 0.12)</td>
</tr>
<tr>
<td>Chidamber Kemerer</td>
<td>-0.44 (p = 0.12)</td>
</tr>
<tr>
<td>Henderson-Sellers</td>
<td>-0.47 (p = 0.06)</td>
</tr>
<tr>
<td>Hitz, Montazeri</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Correlation with expert assessment</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Group 1</td>
</tr>
<tr>
<td>Etzkorn, Gholston, Fortune, Stein, Utley, Farrington, Cox</td>
<td>-0.46 (rating 5/8)</td>
</tr>
<tr>
<td>Chidamber Kemerer</td>
<td>-0.44 (rating 7/8)</td>
</tr>
<tr>
<td>Henderson-Sellers</td>
<td>-0.51 (rating 2/8)</td>
</tr>
<tr>
<td>Hitz, Montazeri</td>
<td></td>
</tr>
</tbody>
</table>
LCC and TCC [Bieman, Kang 1994]

• Recall: LCOM HM “a and b access the same variable”
• What if a calls a’, b calls b’, and a’ and b’ access the same variable?
• Metrics
  • NDP – number of pairs of methods directly accessing the same variable
  • NIP – number of pairs of methods directly or indirectly accessing the same variable
  • NP – number of pairs of methods: \(\frac{n(n-1)}{2}\)
• Tight class cohesion \(TCC = \frac{NDP}{NP}\)
• Loose class cohesion \(LCC = \frac{NIP}{NP}\)
• NB: Constructors and destructors are excluded
## Experimental evaluation of LCC/TCC

<table>
<thead>
<tr>
<th></th>
<th>Correlation with expert assessment</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Group 1</td>
</tr>
<tr>
<td>Etzkorn, Gholston,</td>
<td></td>
</tr>
<tr>
<td>Fortune, Stein, Utley,</td>
<td></td>
</tr>
<tr>
<td>Farrington, Cox</td>
<td></td>
</tr>
<tr>
<td>Chidamber Kemerer</td>
<td>-0.46 (rating 5/8)</td>
</tr>
<tr>
<td>Henderson-Sellers</td>
<td>-0.44 (rating 7/8)</td>
</tr>
<tr>
<td>Hitz, Montazeri</td>
<td>-0.51 (rating 2/8)</td>
</tr>
<tr>
<td>TCC</td>
<td>-0.22 (rating 8/8)</td>
</tr>
<tr>
<td>LCC</td>
<td>-0.54 (rating 1/8)</td>
</tr>
</tbody>
</table>
## Metrics so far…

<table>
<thead>
<tr>
<th>Level</th>
<th>Metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method</td>
<td>LOC, McCabe</td>
</tr>
<tr>
<td>Class</td>
<td>WMC, NOC, DIT, LCOM (and variants), LCC/TCC</td>
</tr>
<tr>
<td>Packages</td>
<td>???</td>
</tr>
</tbody>
</table>
Package metrics

• **Size:**
  - number of classes/interfaces
  - number of classes in the subpackages

• **Dependencies**
  - visualization
  - à la fan-in and fan-out
    - Marchesi’s UML metrics
    - Martin’s $D_n$: abstractness-instability balance or “the normalized distance from the main sequence”
    - PASTA

• Do you still remember aggregations of class metrics?
“Fan-out”

PK₁ or R: 5

Cₑ: 1

[Martin 1994]

3

[JDepend]

4

[Martin 2000]
Fan-in

- “Fan-in” similarly to the “Fan-out”
- Afferent coupling (Martin)
- \( PK_2 \) (Marchesi)

- Dark: TDD, light: no-TDD
- Test-driven development positively affects \( C_a \)
  - The lower \( C_a \) - the better.
- Exception: JUnit vs. Jericho
  - But Jericho is extremely small (2 packages)

[Hilton 2009]
More fan-in and fan-out

- “Fan-in” similarly to the “Fan-out”
- Afferent coupling (Martin)
- PK$_2$ (Marchesi)

<table>
<thead>
<tr>
<th>Marchesi</th>
<th>Man-months</th>
<th>#Pack</th>
<th>avg(PK$_1$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Railway simulator</td>
<td>13</td>
<td>6</td>
<td>8.7</td>
</tr>
<tr>
<td>Warehouse management</td>
<td>7</td>
<td>5</td>
<td>5.0</td>
</tr>
<tr>
<td>CASE tool</td>
<td>13</td>
<td>5</td>
<td>8.1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SAP (Herzig)</td>
<td>Correlation post-release defects</td>
<td>0.091</td>
<td>0.157</td>
<td>0.091</td>
<td>0.084</td>
<td>0.287</td>
<td>0.148</td>
</tr>
</tbody>
</table>
Evolution of afferent and efferent coupling

- Almost all systems show an increasing trend (Lehman’s growing complexity)
- Project 7 (workflow system) is almost stable but very high!
  - Outsourced development
  - No automated tests
  - Severe maintainability problems

Sato, Goldman, Kon 2007
Package metrics: Stability

Stability is related to the amount of work required to make a change [Martin, 2000].

• **Stable** packages
  • Do not depend upon classes outside
  • Many dependents
  • Should be extensible via inheritance (abstract)

• **Instable** packages
  • Depend upon many classes outside
  • No dependents
  • Should not be extensible via inheritance (concrete)
What does balance mean?

A good real-life package must be **instable** enough in order to be easily modified

It must be **generic** enough to be adaptable to evolving requirements, either without or with only minimal modifications

Hence: contradictory criteria
\[ D_n = | \text{Abstractness} + \text{Instability} - 1 | \]

Abstractness = \[ \frac{\#\text{AbstrClasses}}{\#\text{Classes}} \]

Instability = \[ \frac{C_e}{C_e + C_a} \]

Dn – Distance from the main sequence

Normalized distance from the main sequence

- Dark: TDD, light: no-TDD
- Test-driven development positively affects $D_n$
  - The lower $D_n$ - the better.
- The same exception (Jericho vs. JUnit)

[Hilton 2009]
Distribution and evolution

**Exponential distribution**

For all benchmark systems studied, here Vuze 4.0.0.4

**Peak: many feature requests (average Dn)**

![Graph showing distribution and evolution with bars and a line graph]
PASTA [Hautus 2002]

- PASTA – Package structure analysis tool

- Metrics
  - Similarly “fan-in/fan-out”: based on dependencies between packages
  - Go beyond calculating numbers of dependencies

- Focus on dependencies between the subpackages
- Some dependencies are worse than others
  - What are the “bad dependencies”?
  - Cyclic dependencies, layering violations
Idea: remove bad (cycle-causing) dependencies

- **Weight** – number of references from one subpackage to another one.
- Dependencies to be removed are such that
  - The result is acyclic
  - The total weight of the dependencies removed is minimal
- Minimal effort required to resolve all the cycles

Upwards dependencies should be removed
From dependencies to metrics

- **PASTA(P)** = Total weight of the dependencies to be removed / total weight of the dependencies

- No empirical validation of the metrics

- No studies of the metrics evolution

### Table: PASTA Metric

<table>
<thead>
<tr>
<th>Package</th>
<th>PASTA Metric</th>
</tr>
</thead>
<tbody>
<tr>
<td>junit</td>
<td>0%</td>
</tr>
<tr>
<td>org.apache.batik</td>
<td>0%</td>
</tr>
<tr>
<td>org.apache.tools.ant</td>
<td>1%</td>
</tr>
<tr>
<td>java</td>
<td>5%</td>
</tr>
<tr>
<td>org.apache.jmeter</td>
<td>6%</td>
</tr>
<tr>
<td>javax.swing</td>
<td>10%</td>
</tr>
<tr>
<td>org.jboss</td>
<td>11%</td>
</tr>
<tr>
<td>org.gjt.sp.jedit</td>
<td>18%</td>
</tr>
<tr>
<td>java.awt</td>
<td>20%</td>
</tr>
</tbody>
</table>
One metric is good, more metrics are better (?)

• Recall...

\[ MI_1 = 171 - 5.2 \ln(V) - 0.23V(g) - 16.2 \ln(LOC) \]

Halstead  McCabe  LOC

• [Kaur, Singh 2011] propose an adaptation...

\[ MIP = 171 - 5.2CC - 0.23 \ln(S) - 16.2 \ln(NC) \]

Related to PK\textsubscript{1} and instability

Related to NOC and NOM

Related to nesting, strongly connected components, abstractness and PK\textsubscript{2}
Summary: package metrics

- **Size:** number of classes

- **Dependencies à la fan-in and fan-out**
  - Marchesi’s UML metrics
  - Martin’s $D_n$: abstractness-instability balance or “the normalized distance from the main sequence”
  - PASTA

- **Aggregations of class metrics:** reminder
  - Metrics independent: average, sum, Gini/Theil coefficients
  - Metrics dependent: Distribution fitting
Measuring change: Churn metrics

- Why? Past evolution to predict future evolution

- Code Churn [Lehman, Belady 1985]:
  - Amount of code change taking place within a software unit over time

- Code Churn metrics [Nagappan, Bell 2005]:

  **Absolute:**
  - Churned LOC, Deleted LOC, File Count, Weeks of Churn, Churn Count, Files Churned

  **Relative:**
  - M1: Churned LOC / Total LOC
  - M2: Deleted LOC / Total LOC
  - M3: Files churned / File count
  - M4: Churn count / Files churned
  - M5: Weeks of churn / File count
  - M6: Lines worked on / Weeks of churn
  - M7: Churned LOC / Deleted LOC
  - M8: Lines worked on / Churn count
Case Study: Windows Server 2003

• Analyze Code Churn between WS2003 and WS2003-SP1 to predict defect density in WS2003-SP1
  • 40 million LOC, 2000 binaries
  • Use absolute and relative churn measures

• Conclusion 1: Absolute measures are no good
  • \( R^2 < 0.05 \)

• Conclusion 2: Relative measures are good!
  • An increase in relative code churn measures is accompanied by an increase in system defect density
  • \( R^2 \approx 0.8 \)
Case Study: Windows Server 2003

- Construct a statistical model
  - Training set: 2/3 of the Windows Set binaries
- Check the quality of the prediction
  - Test set: remaining binaries
- Three models
  - Right: all relative churn metrics are taken into account
Open issues

- To predict bugs from history, but we need a history filled with bugs to do so
  - Ideally, we don’t have such a history

- We would like to learn from previous projects:
  - Can we make predictions without history?
  - How can we leverage knowledge between projects?
  - Are there universal properties?
  - Not just code properties but also properties of the entire software process
Conclusions

• Package metrics
  • Directly defined: $D_n$, Marchesi metrics, PASTA
  • Results of aggregation

• Churn metrics