2IS55 Software Evolution

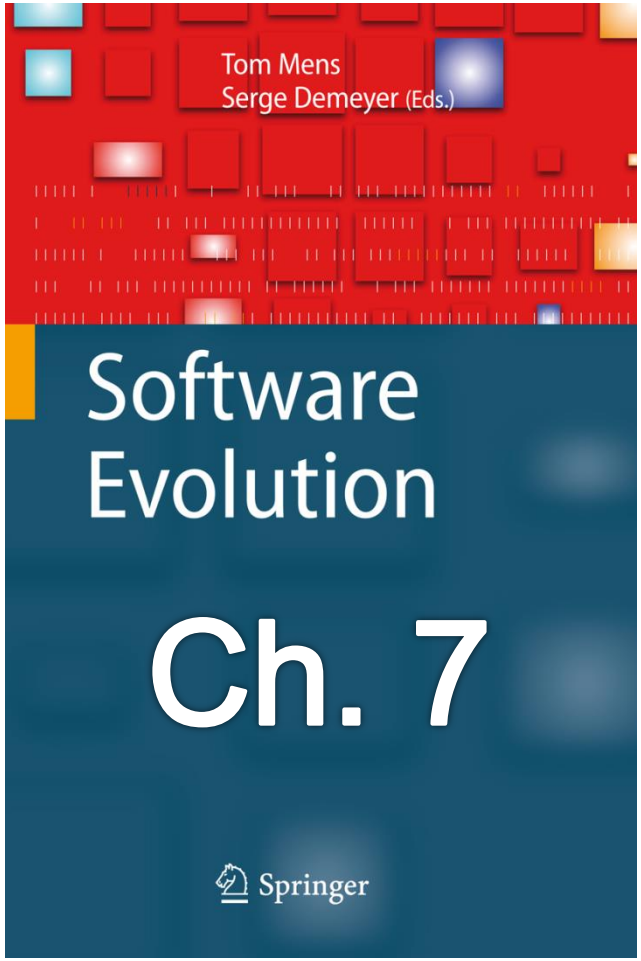
Software metrics (3)

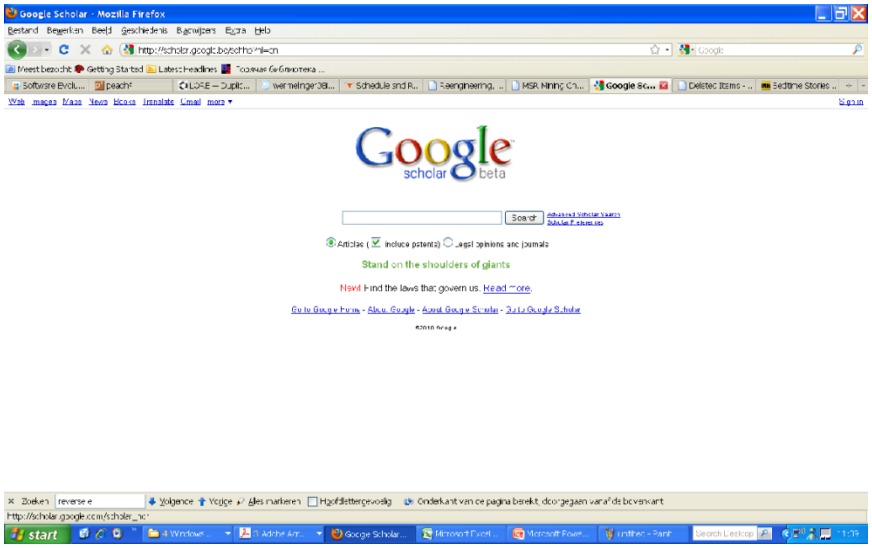
Alexander Serebrenik

Technische Universiteit Eindhoven University of Technology

Where innovation starts

Sources


Tom Mens
Serge Demeyer (Eds.)
Software Evolution
Ch. 7
Springer


Google Scholar - Mozilla Firefox
Google
scholar beta
Stand on the shoulders of giants

From imperative to OO

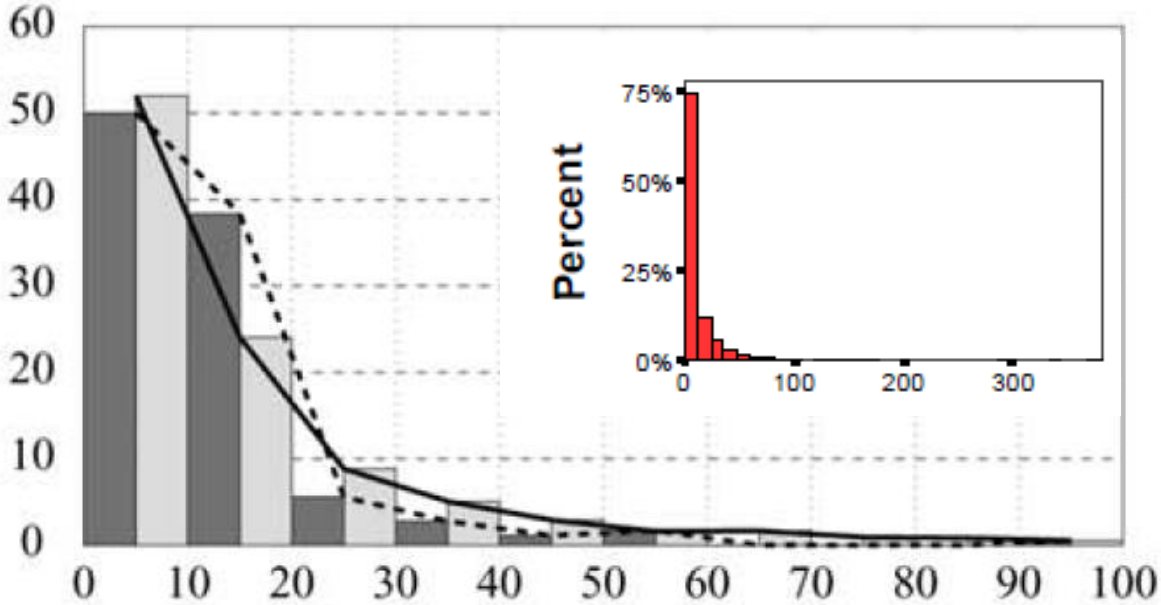
- All metrics so far were designed for imperative languages
  - Applicable for OO
    - On the method level
    - Also
      - Number of files → number of classes/packages
      - Fan-in → afferent coupling (C_a)
      - Fan-out → efferent coupling (C_e)
  - But do not reflect OO-specific complexity
    - Inheritance, class fields, abstractness, …
- Popular metric sets
  - Chidamber and Kemerer, Li and Henry, Lorenz and Kidd, Abreu, Martin

Chidamber and Kemerer

- 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
  - CBO(A) = | {B|A 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

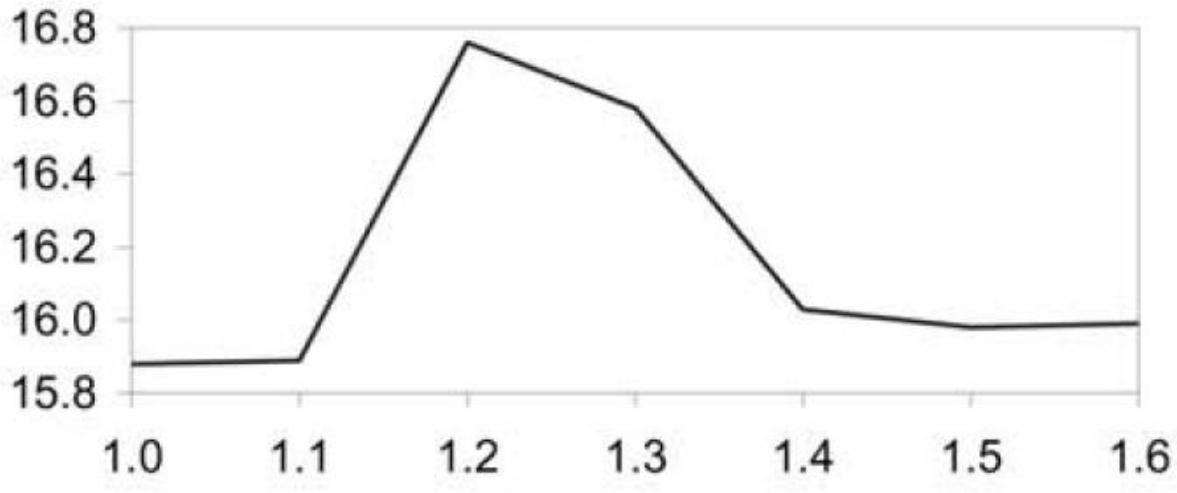
- **WMC – weighted methods per class**
  - **Sum of metrics(m) for all methods m in class C**
  - **Popular metrics: McCabe's complexity and unity**
  - **WMC/unity = number of methods**
  - **Statistically significant correlation with the number of defects**



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

Chidamber and Kemerer

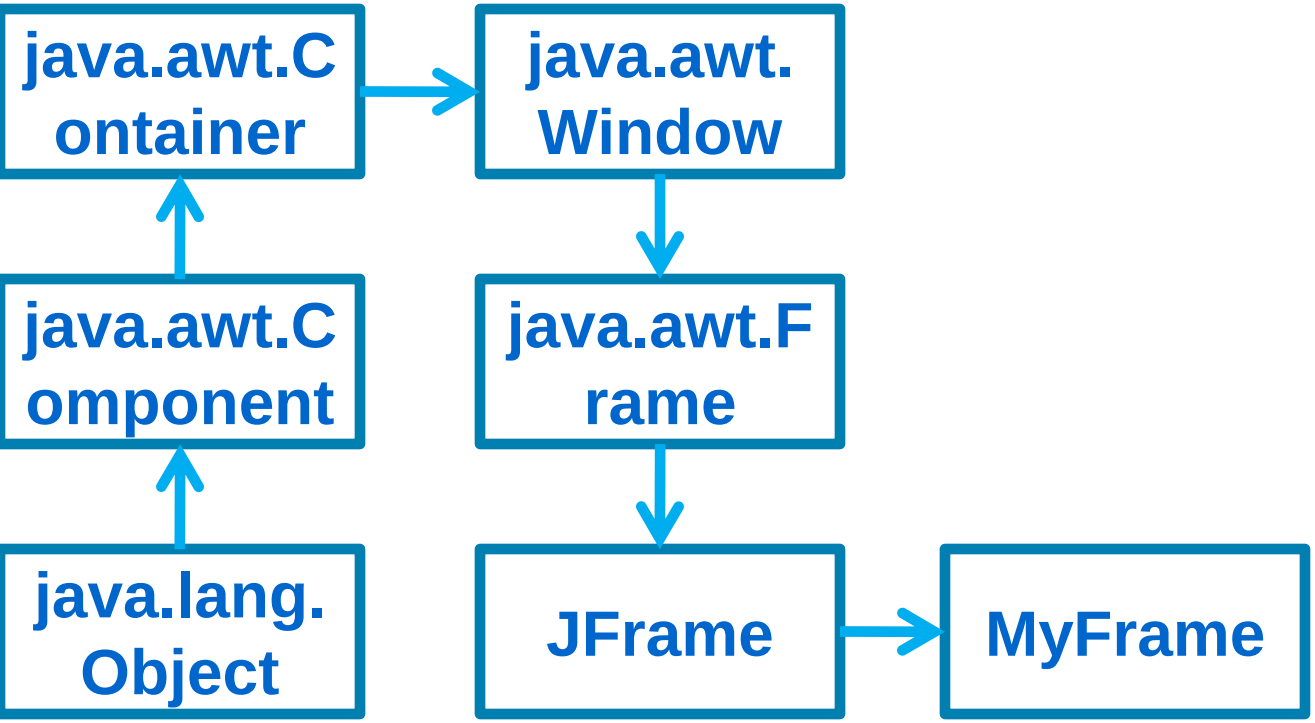
- **WMC – weighted methods per class**
  - **Sum of metrics(m) for all methods m in class C**
  - **Popular metrics: McCabe's complexity and unity**
  - **WMC/unity = number of methods**
  - **Statistically significant correlation with the number of defects**



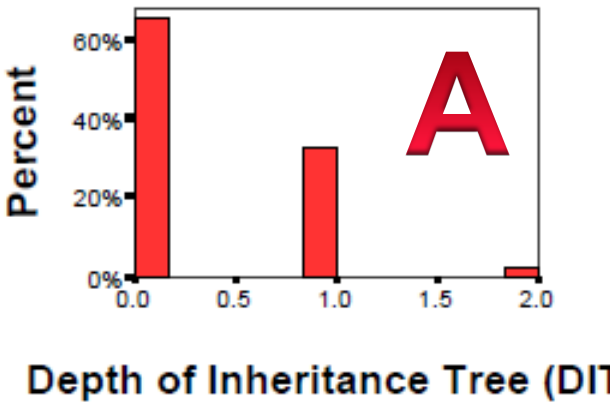
- **WMC/unity**
- **Gyimothy et al.**
- **Average**

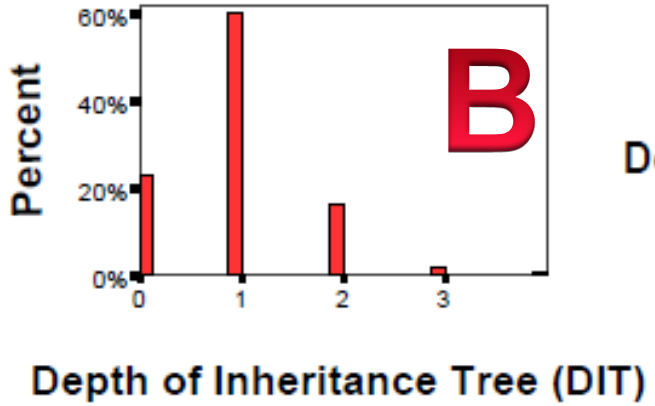
Depth of inheritance - DIT

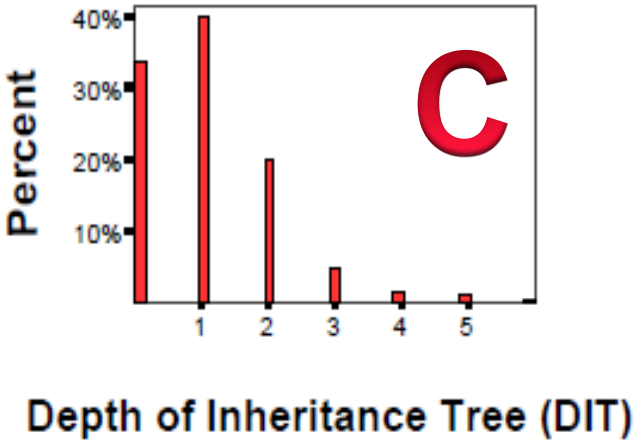
- **Variants: Were to start and what classes to include?**
  - **1, JFrame is a library class, excluded**
  - **2, JFrame is a library class, included**
  - **7**



DIT – what is good and what is bad?

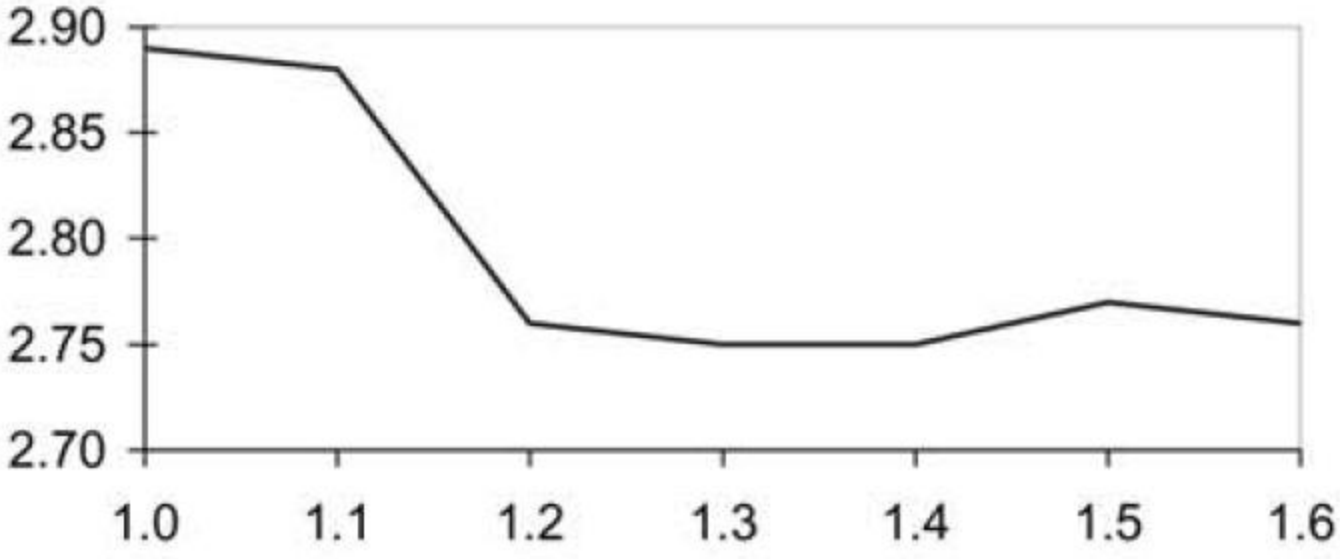






- **Three NASA systems**
- **What can you say about the use of inheritance in systems A, B and C?**
- **Observation: quality assessment depends not just on one class but on the entire distribution**

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

Metric	Our results	[1]	[22]	[21]
WMC	++	+	++	++
DIT	+	++	0	−
RFC	++	++	+	
NOC	0	++	−−	
CBO	++	+	+	+

Significance of CK metrics to predict the number of faults

Modularity metrics: LCOM

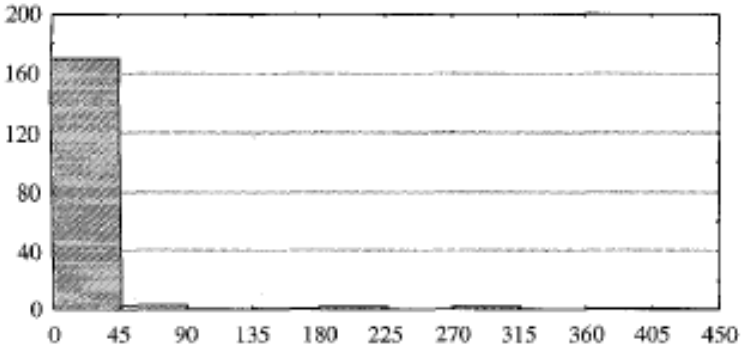
- **LCOM – lack of cohesion of methods**

- **Chidamber Kemerer:**

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

where

- **P = #pairs of distinct methods in C that do not share instance variables**
- **Q = #pairs of distinct methods in C that share instance variables**



[BBM] 180 classes

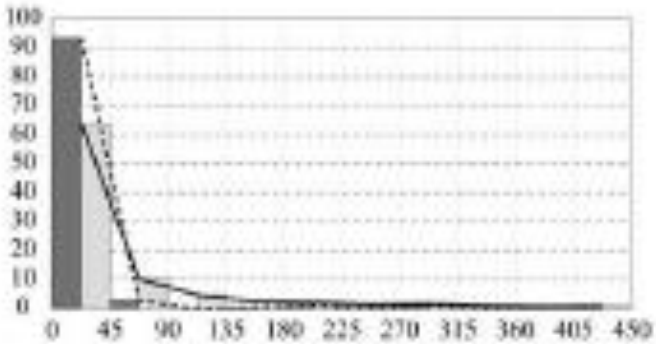
Discriminative ability is insufficient

What about methods that use get/set instead of direct access?

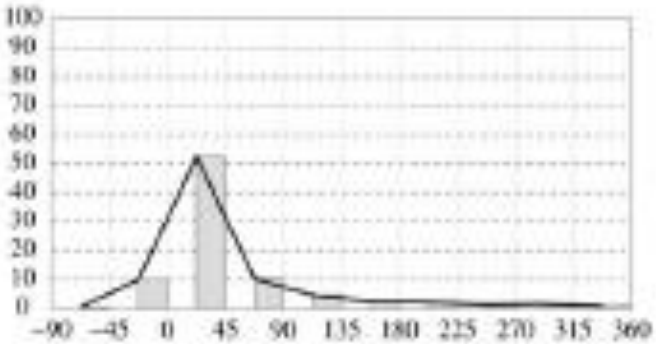
First solution: LCOMN

- **Defined similarly to LCOM but allows negative values**

$$LCOMN(C) = P - Q$$

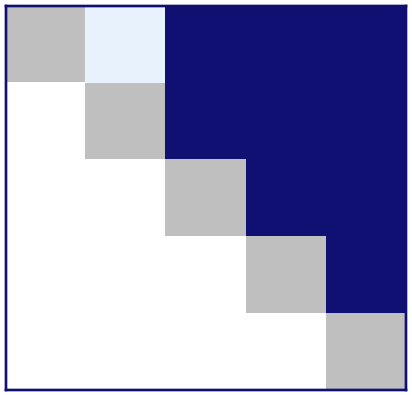


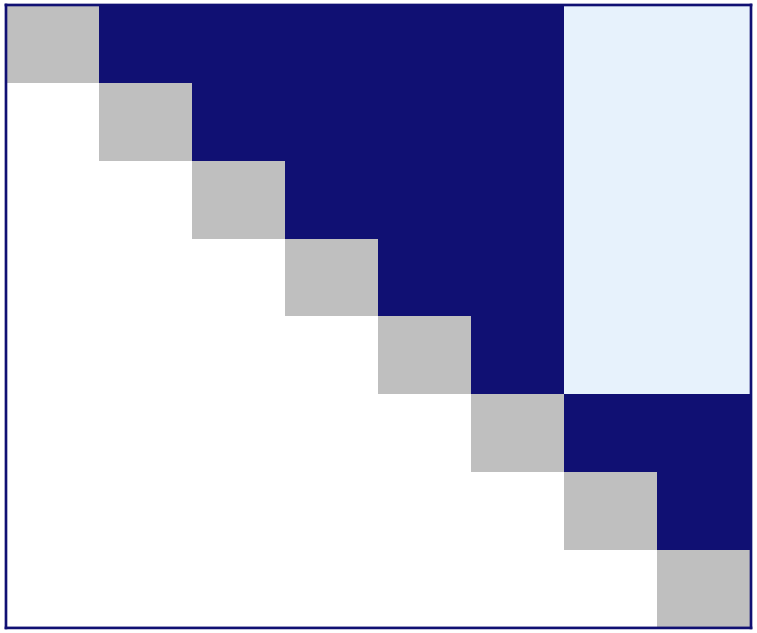
LCOM



LCOMN

Still…





- **Method * method tables**
  - **Light blue: Q, dark blue: P**
- **Calculate the LCOMs**
- **Does this correspond to your intuition?**

Henderson-Sellers, Constantine and Graham 1996

- m – number of methods
- v – number of variables (attrs)
- $m(V_i)$ - #methods that access V_i

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

- 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?

LCOM > 1?

- **If some variables are not accessed at all, then**

$$m(V_i) = 0$$

and if no variables are accessed

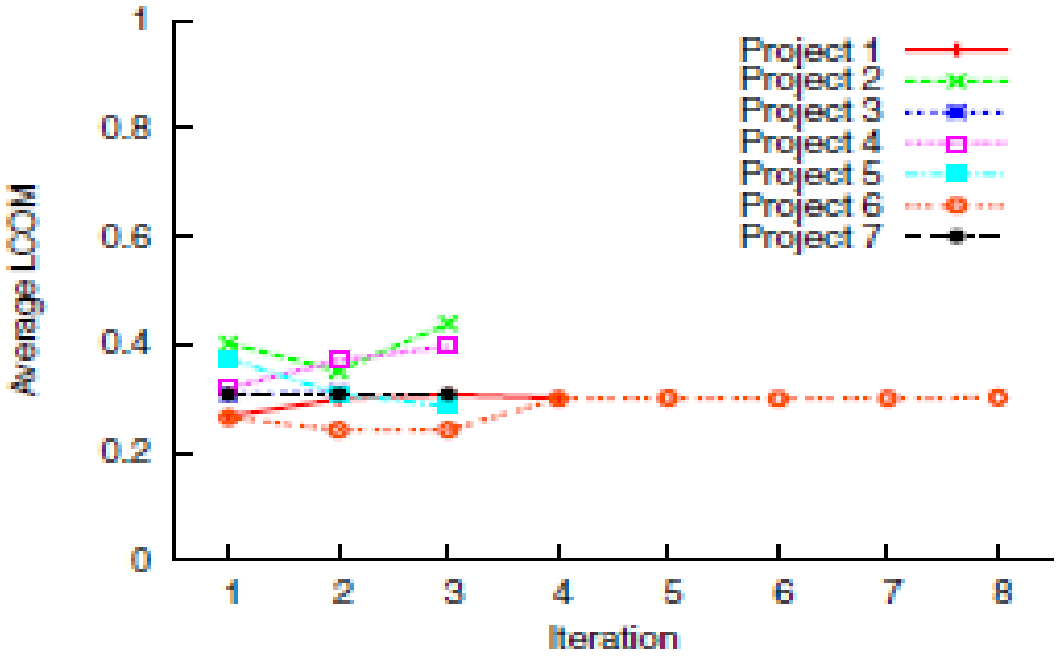
$$\frac{\left(\frac{1}{v}\sum_{i=1}^{v} m(V_i)\right) - m}{1-m} = \frac{-m}{1-m} = 1 + \frac{1}{m-1}$$

Hence

LCOM is undefined for m = 1

LCOM ≤ 2

Evolution of LCOM [Henderson-Sellers et al.]

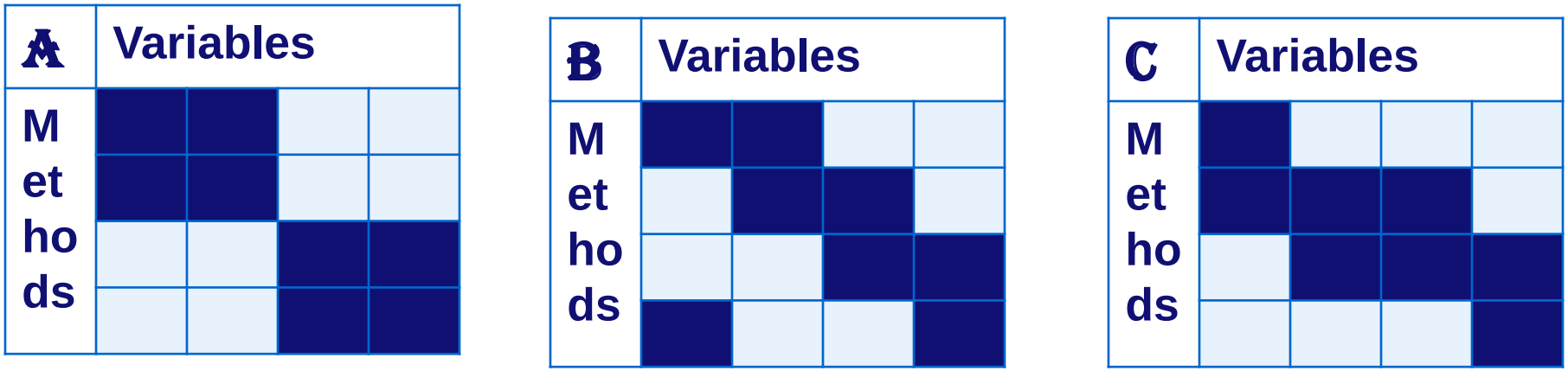


Sato, Goldman, Kon 2007

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

Shortcomings of LCOM [Henderson-Sellers]

Due to [Fernández, Peña 2006]

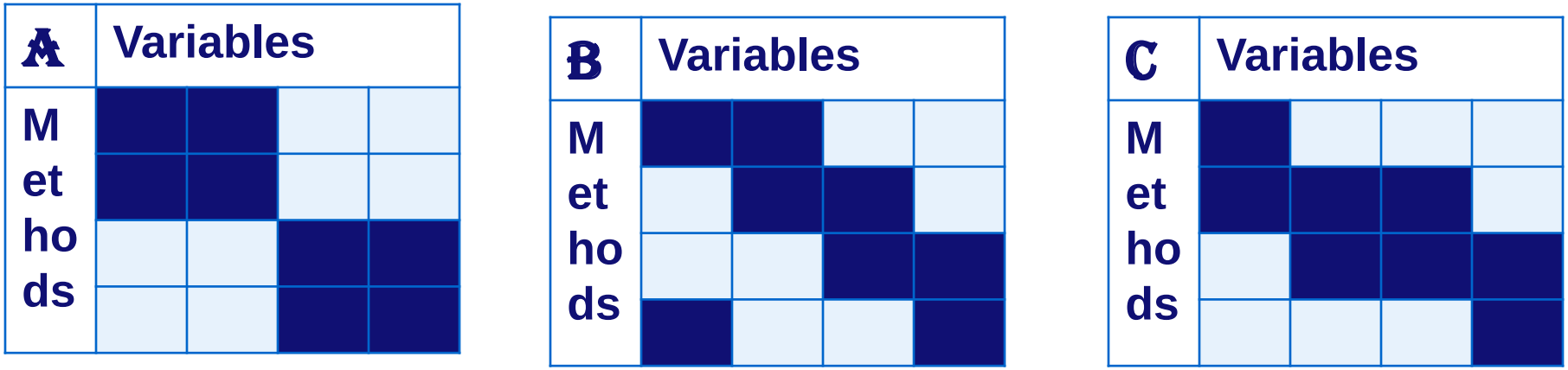


- **Method-variable diagrams: dark spot = access**
- **LCOM(A) ? LCOM(B) ?LCOM(C) ?**

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

Shortcomings of LCOM [Henderson-Sellers]

Due to [Fernández, Peña 2006]



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

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

- **A seems to be less cohesive than B and C!**

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

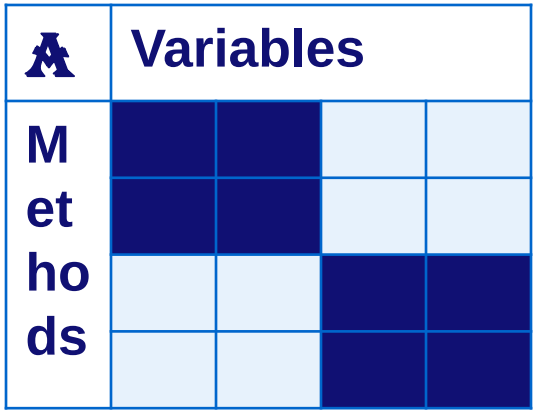
- LCOM values
  - 0, no methods
  - 1, cohesive component
  - 2 or more, lack of cohesion

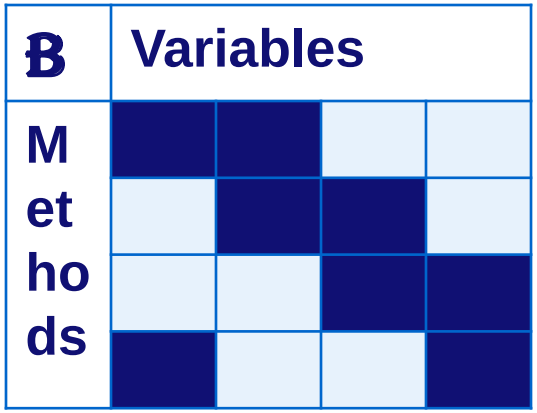
Alternative [Hitz, Montazeri 1995]

- **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**

Question: LCOM?





Experimental evaluation of LCOM variants

Cox, Etzkorn and Hughes 2006	Correlation with expert assessment	
	Group 1	Group 2
Chidamber Kemerer	-0.43 (p = 0.12)	-0.57 (p = 0.08)
Henderson-Sellers	-0.44 (p = 0.12)	-0.46 (p = 0.18)
Hitz, Montazeri	-0.47 (p = 0.06)	-0.53 (p = 0.08)

Etzkorn, Gholston, Fortune, Stein, Utley, Farrington, Cox	Correlation with expert assessment	
	Group 1	Group 2
Chidamber Kemerer	-0.46 (rating 5/8)	-0.73 (rating 1.5/8)
Henderson-Sellers	-0.44 (rating 7/8)	-0.45 (rating 7/8)
Hitz, Montazeri	-0.51 (rating 2/8)	-0.54 (rating 5/8)

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: n(n-1)/2
- Tight class cohesion TCC = NDP/NP
- Loose class cohesion LCC = NIP/NP
- NB: Constructors and destructors are excluded

Experimental evaluation of LCC/TCC

Etzkorn, Gholston, Fortune, Stein, Utley, Farrington, Cox	Correlation with expert assessment	
	Group 1	Group 2
Chidamber Kemerer	-0.46 (rating 5/8)	-0.73 (rating 1.5/8)
Henderson-Sellers	-0.44 (rating 7/8)	-0.45 (rating 7/8)
Hitz, Montazeri	-0.51 (rating 2/8)	-0.54 (rating 5/8)
TCC	-0.22 (rating 8/8)	-0.057 (rating 8/8)
LCC	-0.54 (rating 1/8)	-0.73 (rating 1.5/8)

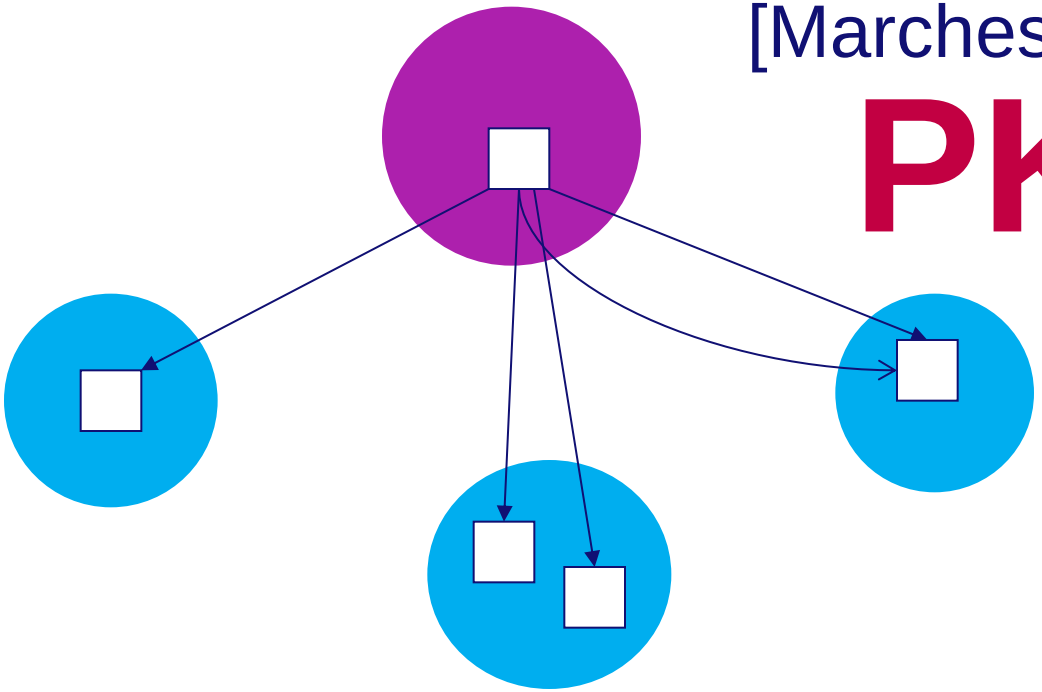
Metrics so far…

Level	Metrics
Method	LOC, McCabe
Class	WMC, NOC, DIT, LCOM (and variants), LCC/TCC
Packages	???

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”



[Marchesi 1998] [Martin 2000]

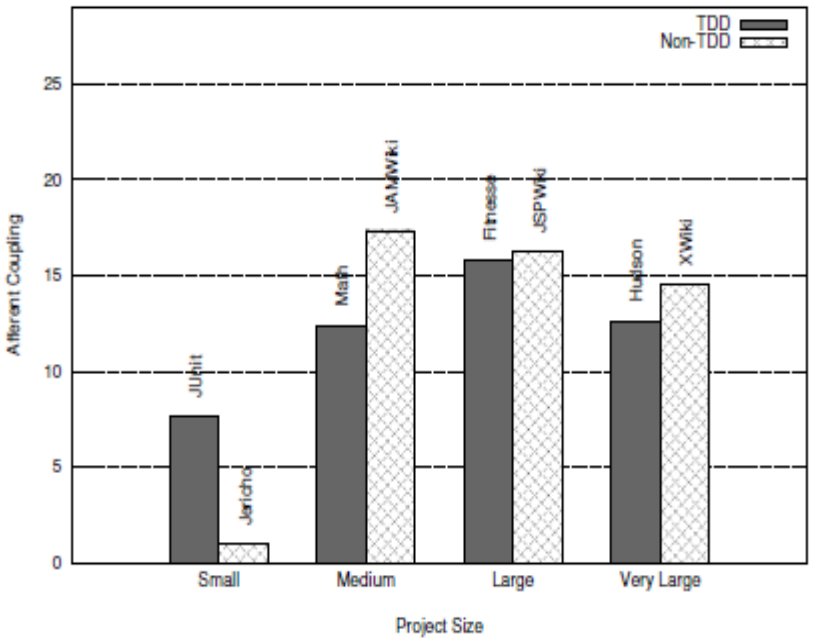
PK_1 or R: 5

C_e: 1 3 4

[Martin 1994] [JDepend] [Martin 2000]

Fan-in

- "Fan-in" similarly to the "Fan-out"
  - Afferent coupling (Martin)
  - PK_2 (Marchesi)



[Hilton 2009]

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

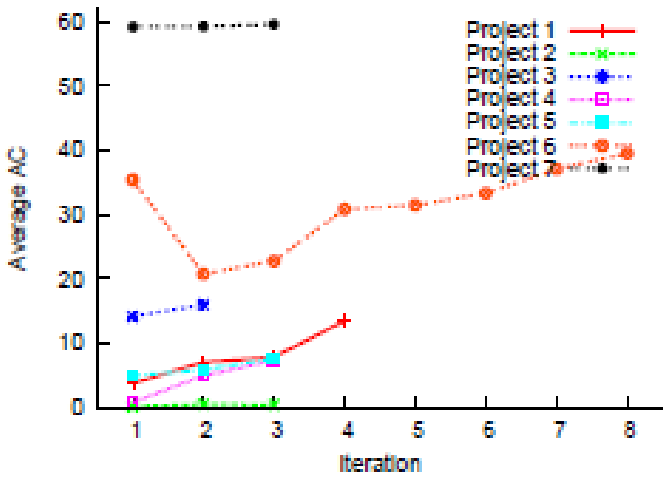
More fan-in and fan-out

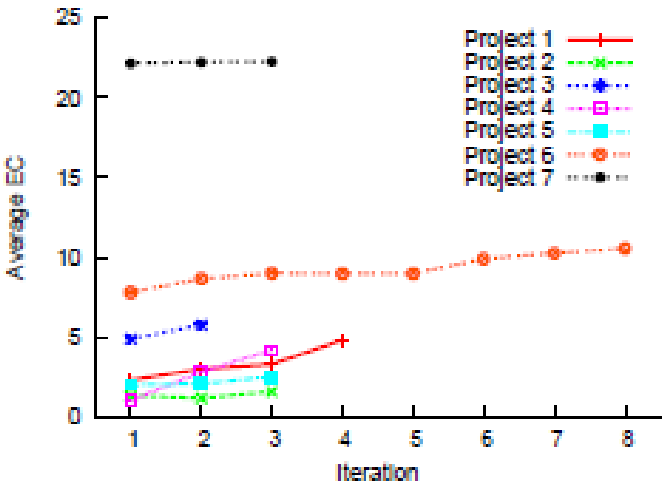
- "Fan-in" similarly to the "Fan-out"
  - Afferent coupling (Martin)
  - PK_2 (Marchesi)

SAP (Herzig)	Correlation post-release defects
Afferent	0.091
Efferent [Martin 2000]	0.157
Class-in	0.084
Class-out	0.086
Fan-in	0.287
Fan-out	0.148

Marchesi	Man-months	#Pack	avg(PK_1)
Railway simulator	13	6	8.7
Warehouse management	7	5	5.0
CASE tool	13	5	8.1

Evolution of afferent and efferent coupling





Sato, Goldman, Kon 2007

- **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**

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

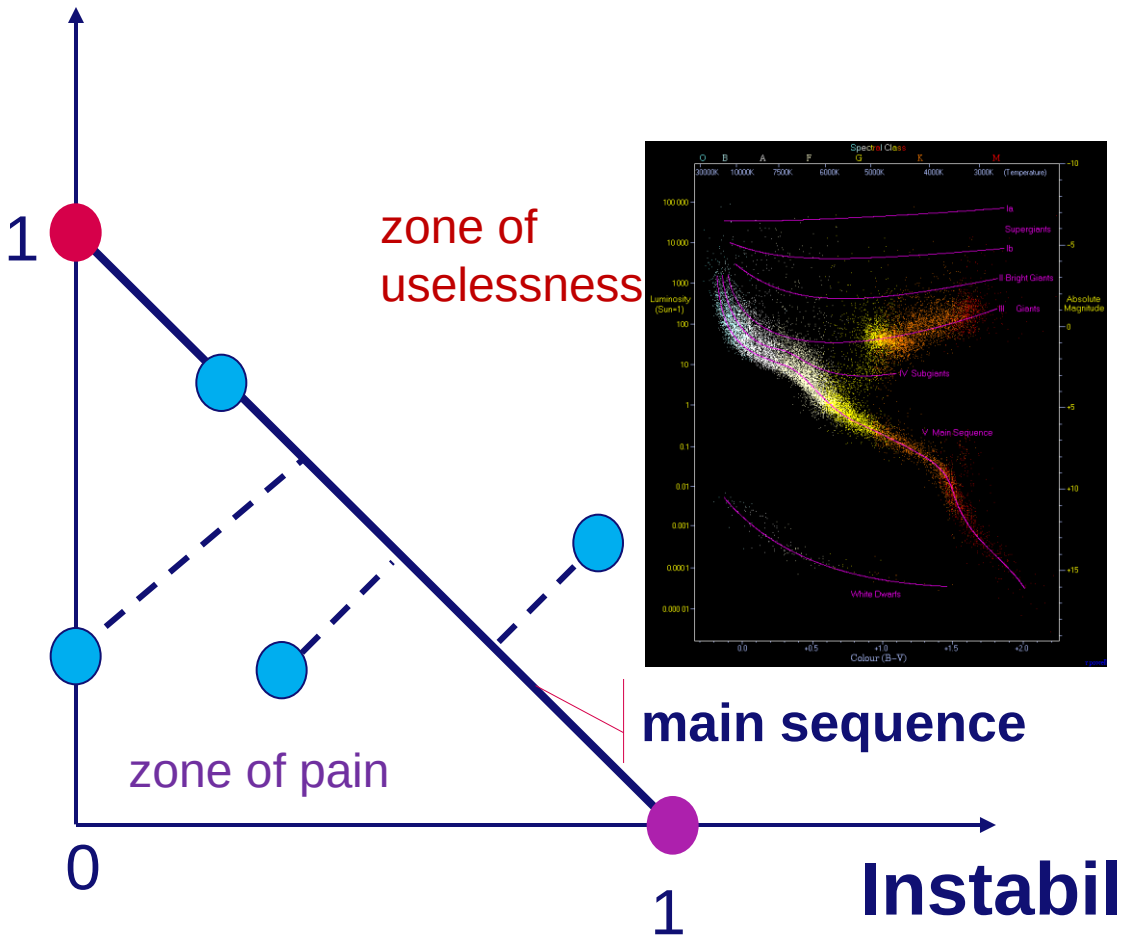
It must be **generic** enough to be adaptable to evolving requirements, either without or with only minimal modifications

Hence: contradictory criteria

D_n – Distance from the main sequence

Abstractness =

#AbstrClasses/#Classes



D_n =

| Abstractness +

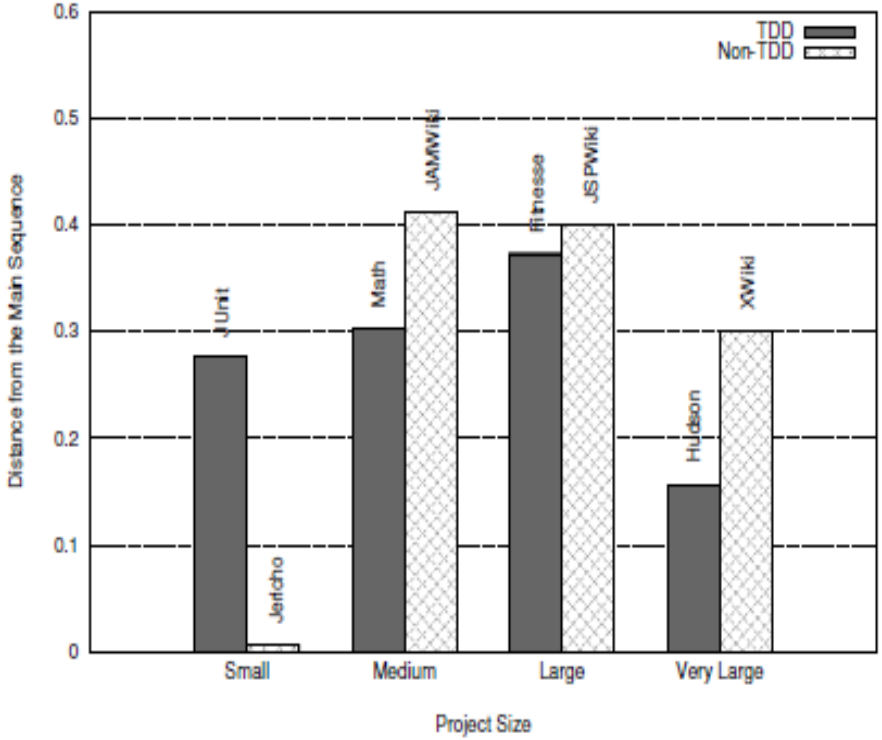
Instability – 1 |

[R.Martin 1994]

Instability =

$C_e/(C_e+C_a)$

Normalized distance from the main sequence



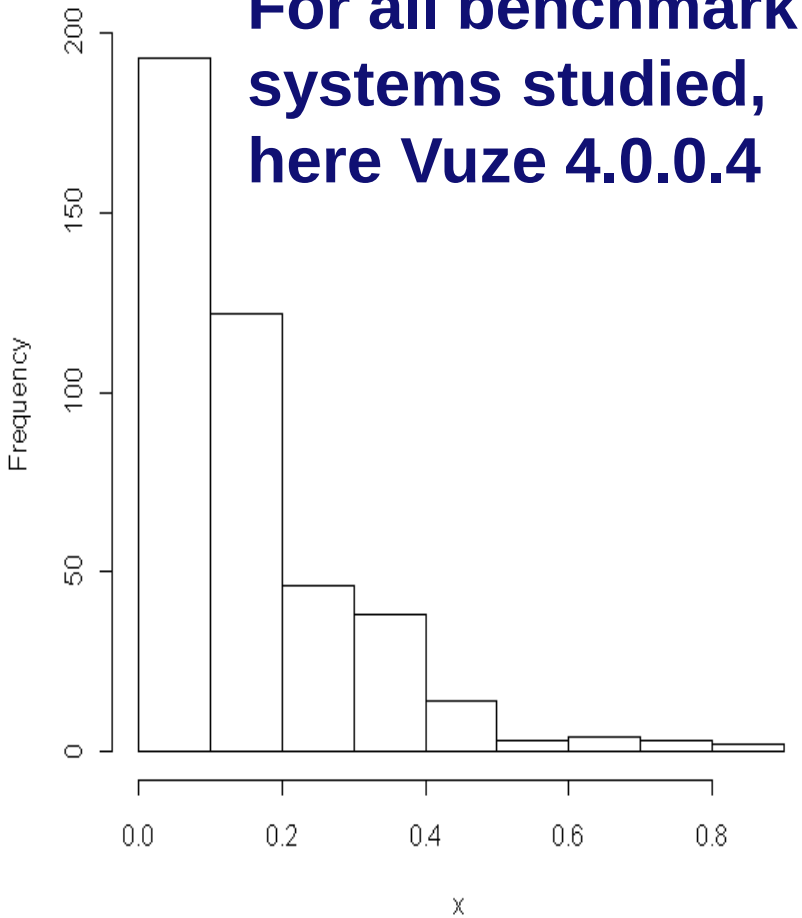
[Hilton 2009]

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

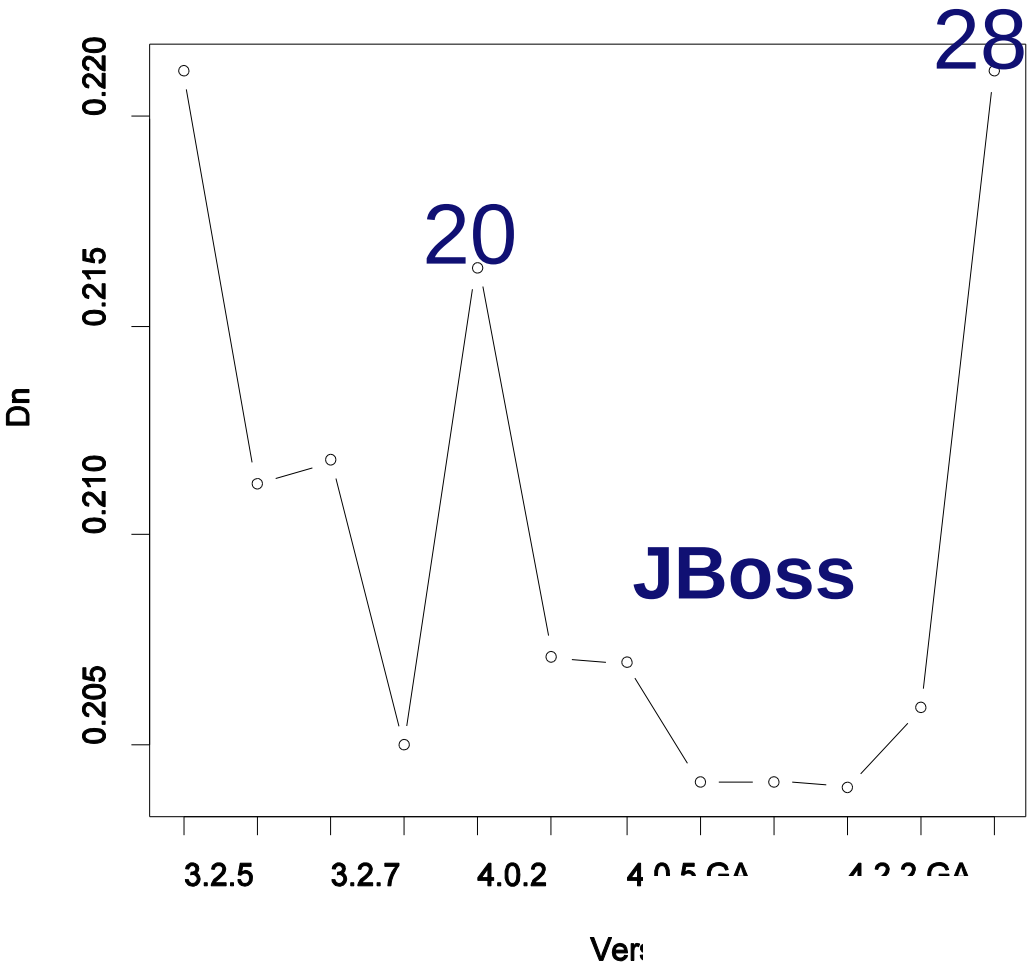
Distribution and evolution

Exponential distribution

For all benchmark systems studied, here Vuze 4.0.0.4



Peak: many feature requests (average Dn)



PASTA [Hautus 2002]

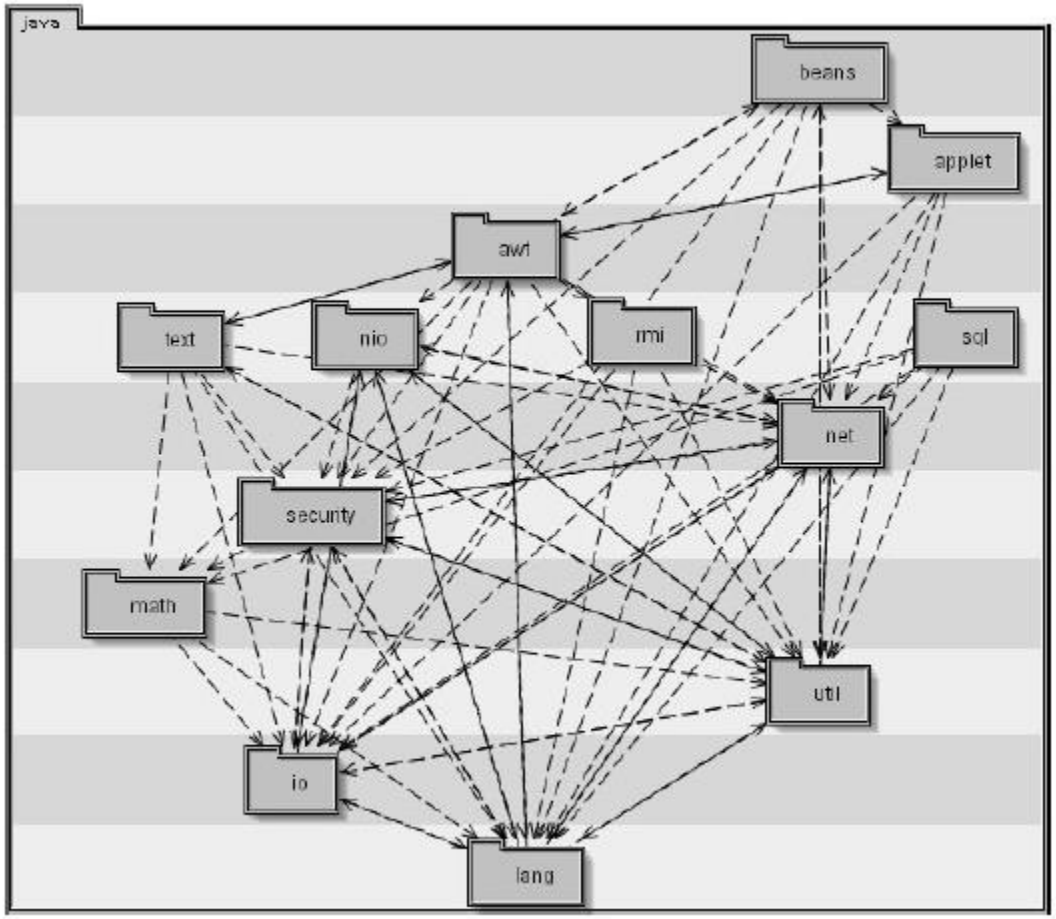
- PASTA – Package structure analysis tool

- 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

PASTA [Hautus]

- **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**

Package	PASTA Metric
junit	0%
org.apache.batik	0%
org.apache.tools.ant	1%
java	5%
org.apache.jmeter	6%
javax.swing	10%
org.jboss	11%
org.gjt.sp.jedit	18%
java.awt	20%

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_1 and instability

Related to NOC and NOM

Related to nesting, strongly connected components, abstractness and PK_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

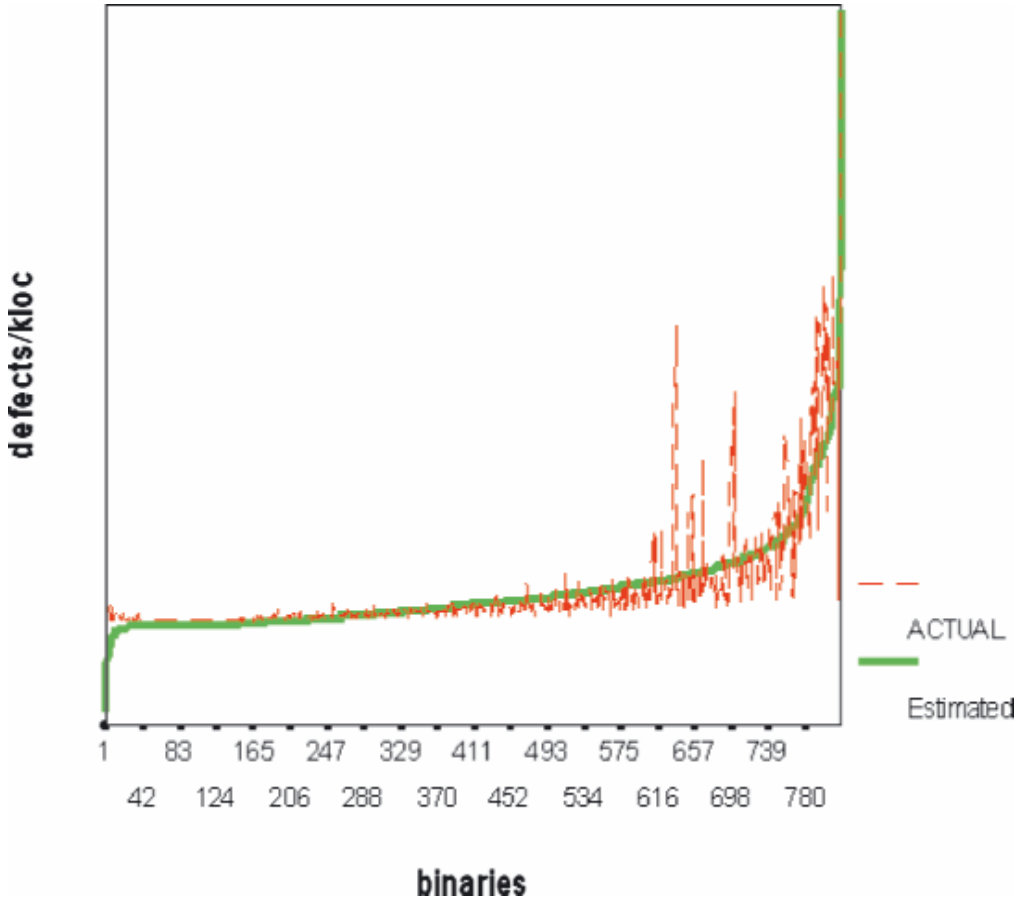
- **Analyze Code Churn between WS2003 and WS2003-SP1 to predict defect density in WS2003-SP1**
  - **40 million LOC, 2000 binaries**
  - **Use absolute and relative churn measures**

- **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**