2IS55 Software Evolution

Software metrics

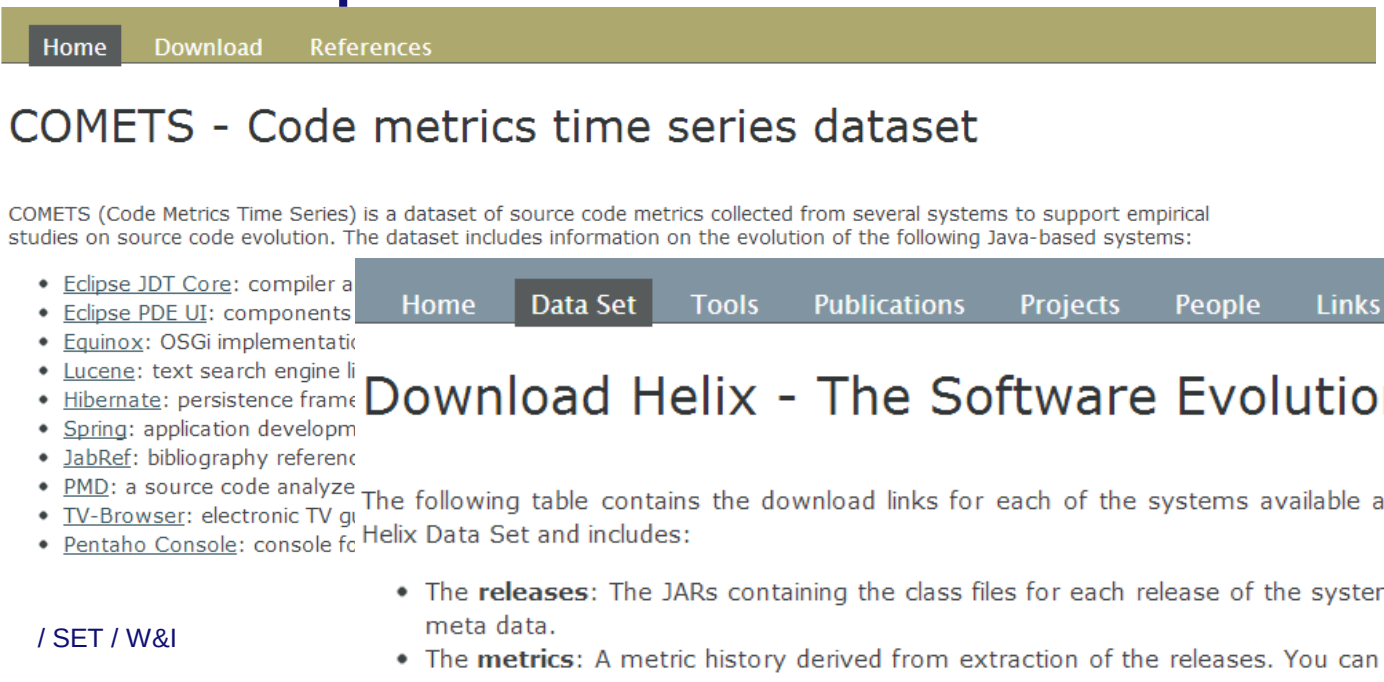
Alexander Serebrenik

TU/e Technische Universiteit Eindhoven University of Technology

Where innovation starts

Assignments

- **Assignment 3: deadline – March 17!**
- **Assignment 4: testing**
- **Assignment 5:**
  - **Do software evolution laws hold in practice?**
  - **Use software metrics to instrument and verify the claim**
  - **Preprocessed datasets**



SWINBURNE UNIVERSITY OF TECHNOLOGY

Recall…

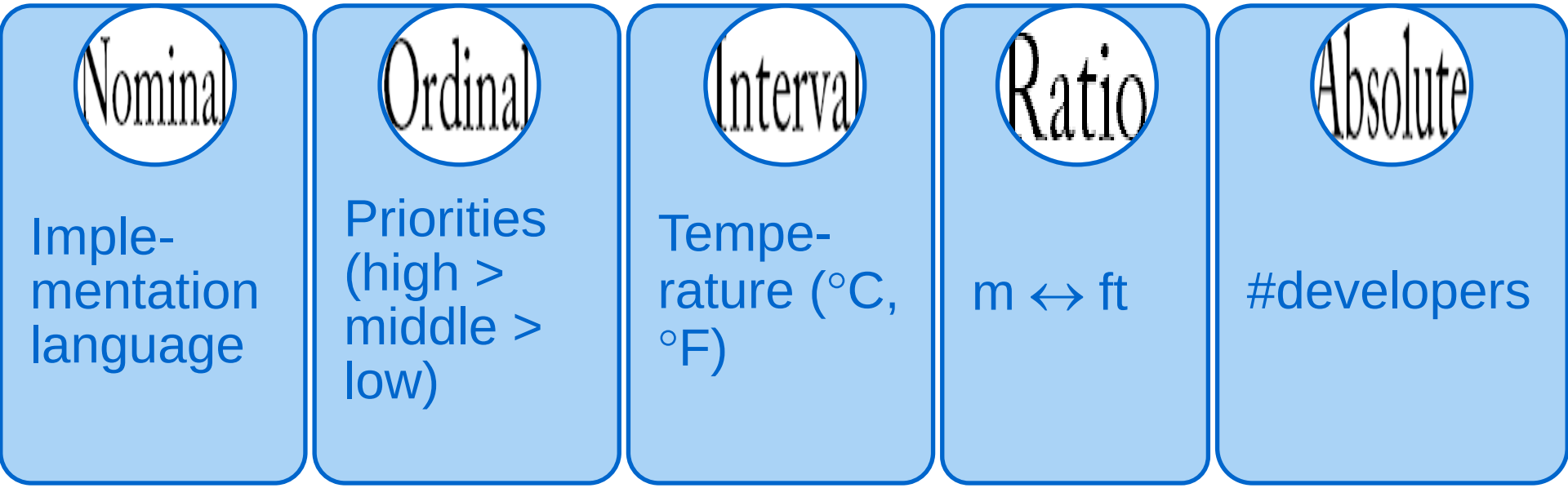
- **Metric:**
  - **“A quantitative measure of the degree to which a system, component, or process possesses a given variable. ” --- IEEE Standard 610.12-1990**
  - **“A software metric is any type of measurement which relates to a software system, process or related documentation.” --- Ian Sommerville, Software Eng. 2006**

  - **Short: mapping of software artefacts to a well-known domain**

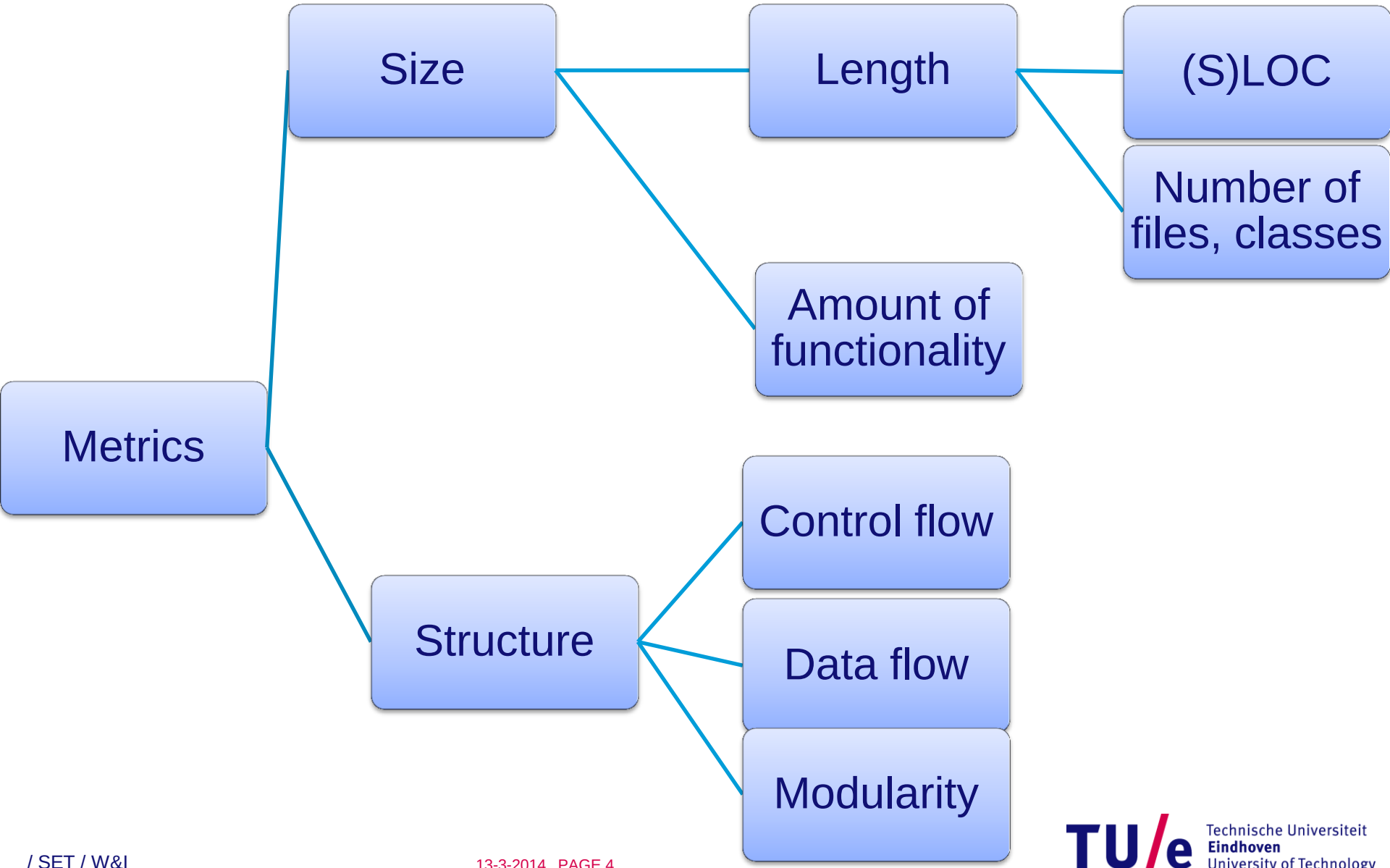
Metrics and scales

- **What metrics have we seen so far?**
  - **Size: LOC, SLOC**
  - **Code duplication: POP, RNR, …**
  - **Requirements: Flesch-Kincaid grade level**

To what scale does it belong?



Classification of metrics [à la Fenton, Pfleeger 1996]



Program length (LOC)

- **Variants:**
  - **Total**
  - **Non-blank**
  - **SLOC (source LOC): Ignore comments and blank lines**
  - **LLOC (logical LOC): Number of program statements**

```
1   for (i = 0;
2            i < 100;
3                i += 1) {
4        printf("hello");
5   }
6
7   /* An important loop */
```

Total LOC: 7
Non-blank LOC: 6
SLOC: 5
LLOC: 2 (for and printf)

Advantages of (S)LOC

- Related to Lehman's law of "continuous growth" (Law 6)
- Easy to calculate
  - LLOC is more difficult to determine (parser needed)
  - What happens with nested statements? for(i=0;i<10;i++)?
- Correlation with the #bugs
  - Moderate (0.4-0.5) [Rosenberg 1997, Zhang 2009]
  - Larger modules usually have more bugs
    - "Ranking ability of LOC" [Fenton and Ohlsson 2000 , Zhang 2009]
  - There are better (but more complex) ways to predict #bugs
  - Can be used to predict the development effort!

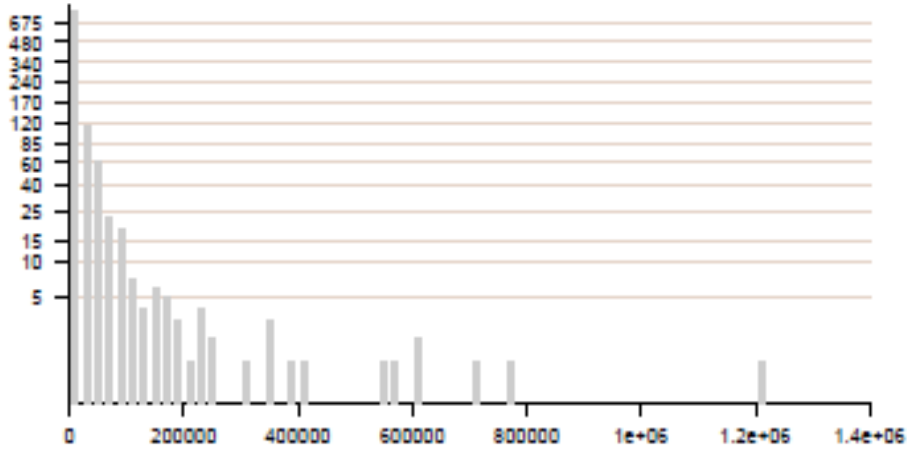
Disadvantages of (S)LOC

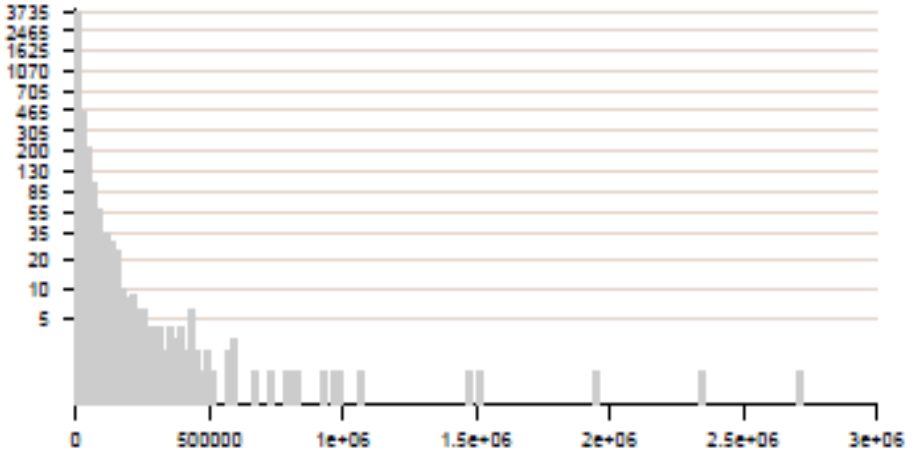
- Ignores structure of the program
  - Program code is more than just text!

- Difficult to compare modules in different languages or written by different developers
  - Some languages are more verbose due to
    - Presence/absence of “built-in” functionality
    - Structural verbosity (e.g., .h in C)
  - Some developers are paid per LOC!
  - Hand-written vs. generated code

(S)LOC distribution

Robles et al. 2006





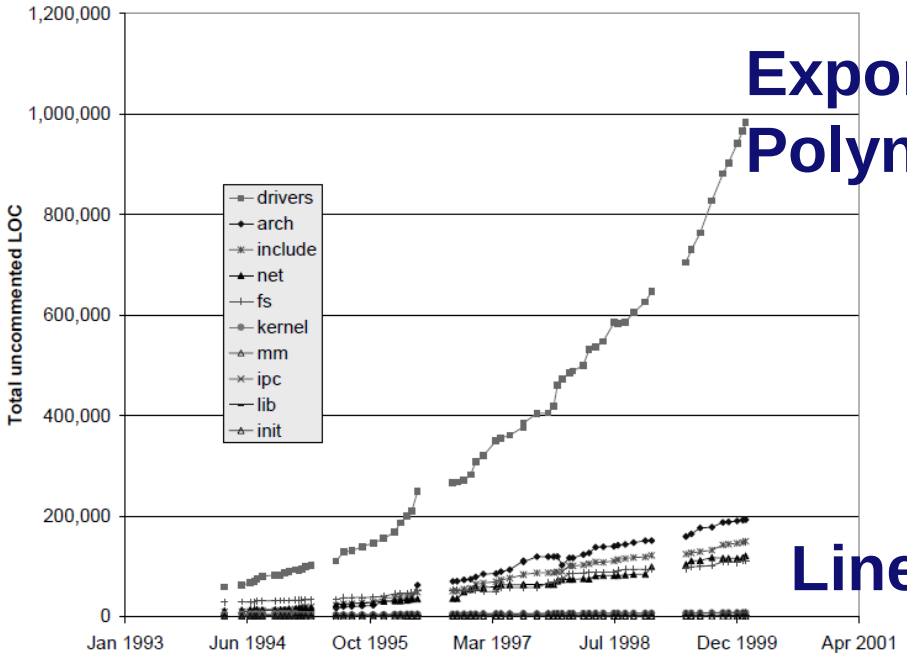
- Distribution of SLOC in Debian 2.0 (left) and 3.0 (right)
- Controversy: log-normal or double Pareto?
  - Importance: knowing distribution one can estimate the probability to obtain files of a given size
  - Hence, to estimate size of the entire system
  - And the effort required (size ⇒ effort)

What do we know about evolution of SLOC?

- **Related to Lehman's 6:**
  - **The functional capability <…> must be continually enhanced to maintain user satisfaction over system lifetime.**
  - **Earlier versions: "size".**

- **Also related to Lehman's 5:**
  - **In general, the incremental growth (growth rate trend) of E-type systems is constrained by the need to maintain familiarity.**
  - **Lehman interpreted this as linear growth**

What do we know about evolution of SLOC?



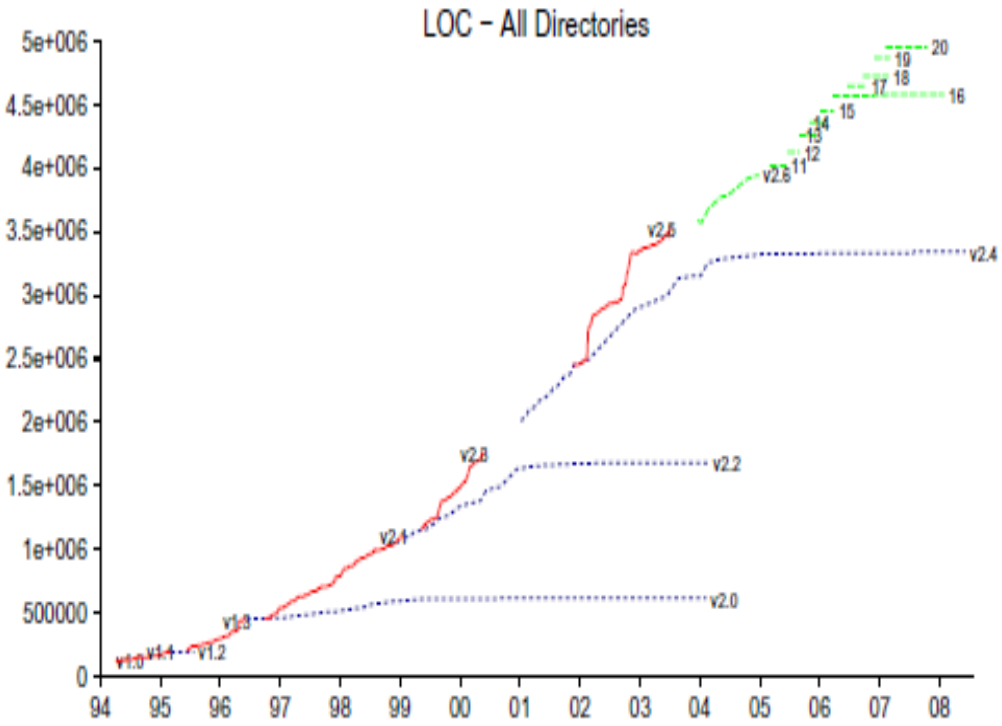
Exponential?

Polynomial?

Linear

- **Godfrey and Tu: superlinear growth is typical for OS**
  - **Koch 2007: Quadratic growth is better for larger OS projects (study of 8621 OS projects on SourceForge)**

LOC in Linux kernel



Superlinear up to 2.5, linear for 2.6

Scacchi – mix of superlinear and sublinear

Israeli, Feitelson:

- **Linux kernel**
- **Multiple versions and variants**
  - **Production (blue dashed)**
  - **Development (red)**
  - **Current 2.6 (green)**

(S)LOC: Summary

- Different variants: LOC, SLOC, LLOC
- Advantages:
  - Easy to compute, moderately correlates with #bugs
  - Can be used to estimate the development effort (more details on May 15)
- Disadvantages
  - Different programming languages and developers
  - Hand-written vs. generated code
- Distribution “exponential-like”
- Evolution:
  - Linear
  - Linux (other OS?): Superlinear
  - Mix

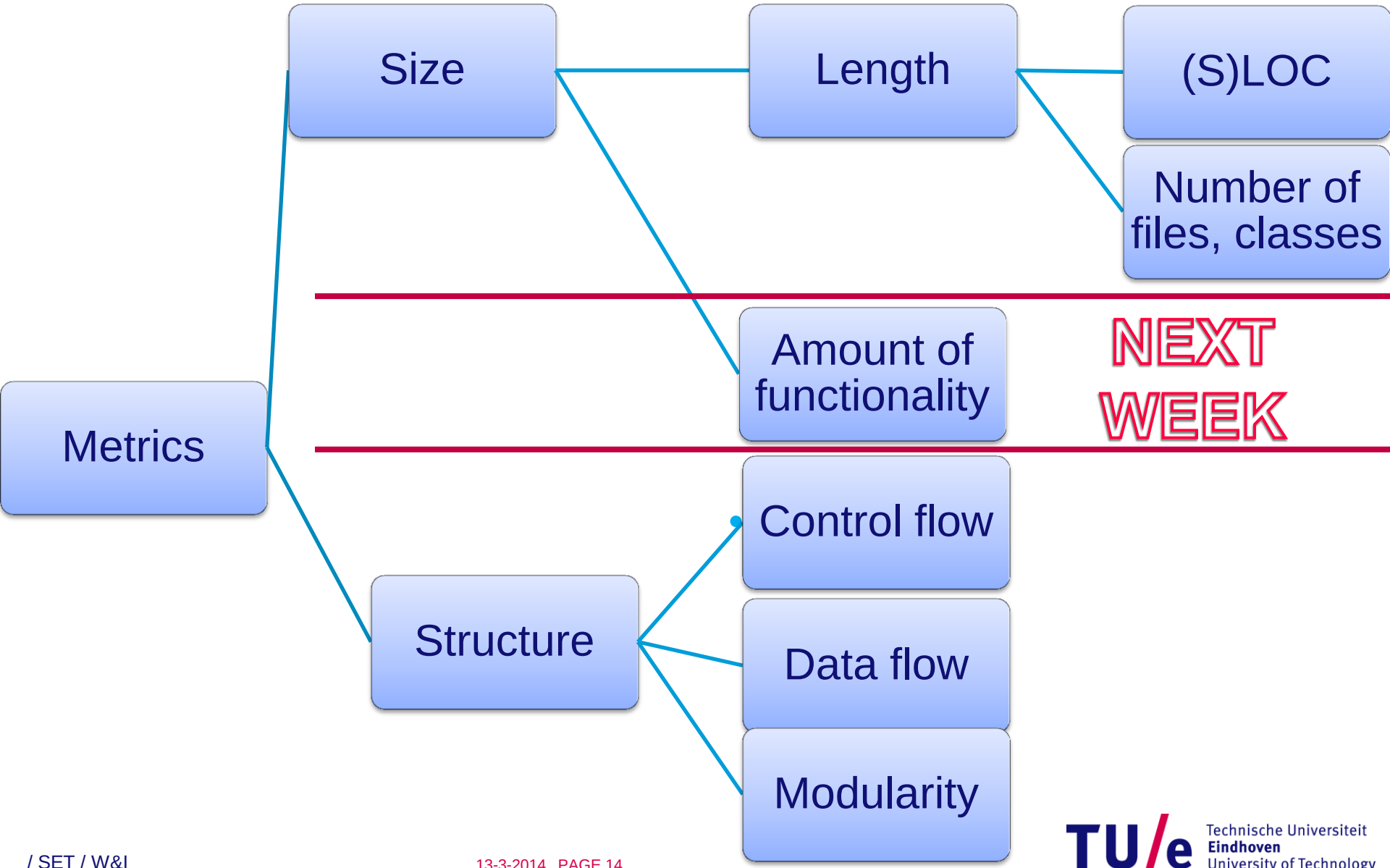
Length: #components

- Number of files, classes, packages
- Intuitive: “number of volumes in an encyclopaedia”

- Variants:
  - All files, classes, packages
  - No empty/library/third-party files, classes, packages
  - No nested/inner classes
  - No or only some auxiliary files (makefiles, header files)

- Correlation with the #post-release defects [Nagappan, Ball, Zeller 2006]
  - significant for modules A, B, C (strength:0.5-0.7), insignificant for modules D, E
  - for each module correlation with some other metrics!

So far


Size
Length
(S)LOC
Number of files, classes
Amount of functionality
NEXT WEEK
Metrics
Structure
Control flow
Data flow
Modularity

Complexity metrics: Halstead (1977)

- Sometimes is classified as size rather than complexity
- Unit of measurement

Parts of a statement: operators and operands

Line: LOC, SLOC, LLOC

Units, files, classes

Packages, directories

- Operators:
  - traditional (+,++, >), keywords (return, if, continue)
- Operands
  - identifiers, constants

Halstead metrics

- Four basic metrics of Halstead

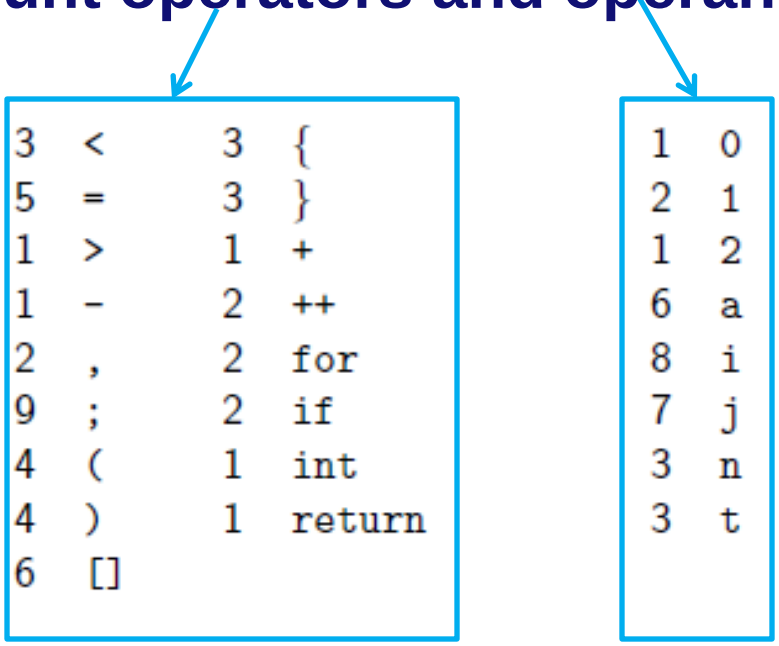
	Total	Unique
Operators	N1	n1
Operands	N2	n2

- Length: N = N1 + N2
- Vocabulary: n = n1 + n2
- Volume: $V = N \log_2 n$
  - Insensitive to lay-out
  - VerifySoft:
    - $20 \leq$ Volume(function) ≤ 1000
    - $100 \leq$ Volume(file) ≤ 8000

Halstead metrics: Example

```
void sort ( int *a, int n ) {
int i, j, t;

   if ( n < 2 ) return;
   for ( i=0 ; i < n-1; i++ )  {
         for ( j=i+1 ; j < n ; j++ ) {
               if ( a[i] > a[j] ) {
                     t = a[i];
                     a[i] = a[j];
                     a[j] = t;
               }
         }
   }
}
```

- Ignore the function definition
- Count operators and operands



Operators		Operands	
3	<	3	{
5	=	3	}
1	>	1	+
1	-	2	++
2	,	2	for
9	;	2	if
4	(	1	int
4	)	1	return
6	[]		

1	0
2	1
1	2
6	a
8	i
7	j
3	n
3	t

	Total	Unique
Operators	N1 = 50	n1 = 17
Operands	N2 = 30	n2 = 7

$V = 80 \log_2(24) \approx 392$

Inside the boundaries [20;1000]

Further Halstead metrics

	Total	Unique
Operators	N1	n1
Operands	N2	n2

- Volume: $V = N \log_2 n$
- Difficulty: D = (n1 / 2) * (N2 / n2)
  - Sources of difficulty: new operators and repeated operands
  - Example: 17/2 * 30/7 ≈ 36
- Effort: E = V * D
- Time to understand/implement (sec): T = E/18
  - Running example: 793 sec ≈ 13 min
  - Does this correspond to your experience?
- Bugs delivered: $E^{2/3}/3000$
  - For C/C++: known to underapproximate
  - Running example: 0.19

Halstead metrics are sensitive to…

- What would be your answer?

- Syntactic sugar:

i = i+1	Total	Unique
Operators	N1 = 2	n1 = 2
Operands	N2 = 3	n2 = 2

i++	Total	Unique
Operators	N1 = 1	n1 = 1
Operands	N2 = 1	n2 = 1

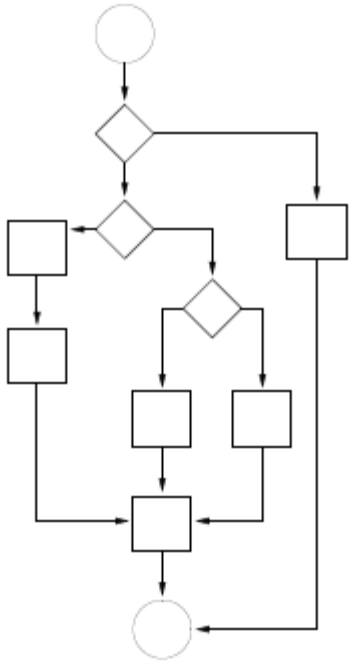
- Solution: normalization (see the code duplication slides)

Structural complexity

- **Structural complexity:**
  - **Control flow**
  - **Data flow**

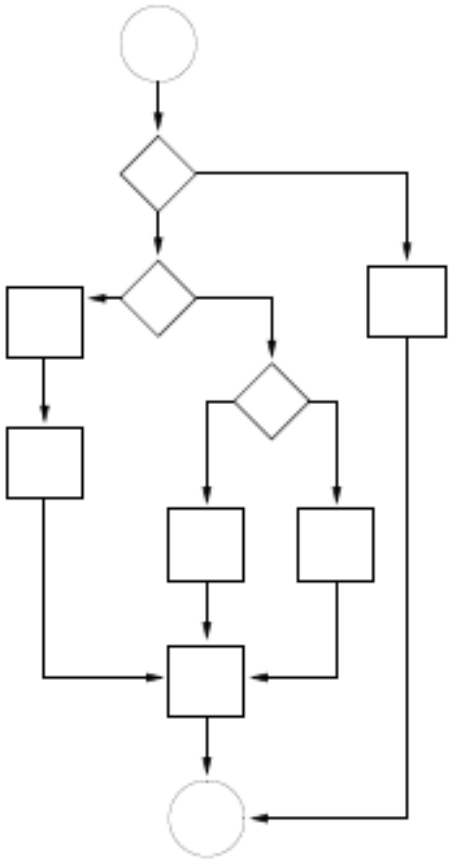
  } **Commonly represented as graphs** → **Graph-based metrics**

  - **Modularity**



- **Number of vertices**
- **Number of edges**
- **Maximal length (depth)**

McCabe's complexity (1976)



In general

- **v(G) = #edges - #vertices + 2**

For control flow graphs

- **v(G) = #binaryDecisions + 1, or**
- **v(G) = #IFs + #LOOPs + 1**

Number of paths in the control flow graph.

A.k.a. "cyclomatic complexity"

Each path should be tested!

v(G) – a testability metrics

Boundaries
- **v(function) ≤ 15**
- **v(file) ≤ 100**

McCabe's complexity: Example

```
void sort ( int *a, int n ) {
int i, j, t;

   if ( n < 2 ) return;
   for ( i=0 ; i < n-1; i++ )  {
         for ( j=i+1 ; j < n ; j++ ) {
               if ( a[i] > a[j] ) {
                     t = a[i];
                     a[i] = a[j];
                     a[j] = t;
               }
         }
   }
}
```

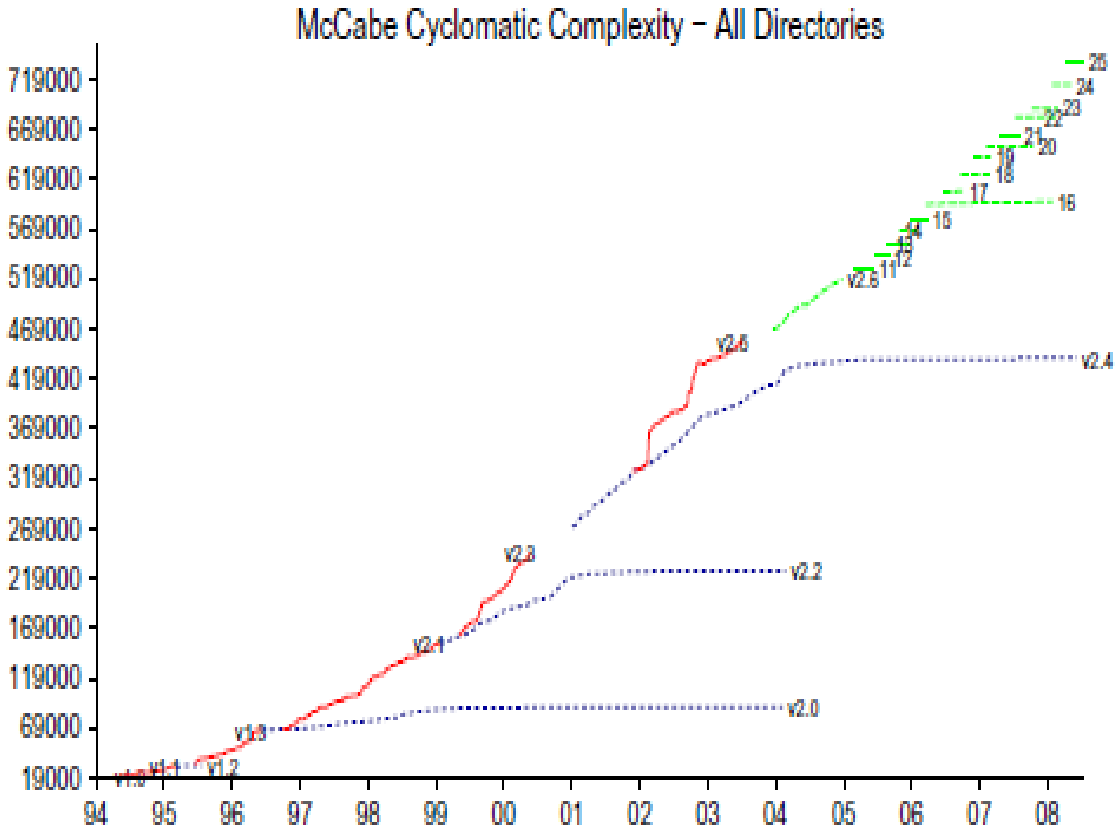
- Count IFs and LOOPs
- IF: 2, LOOP: 2
- v(G) = 5
- Structural complexity

Question to you

- **Is it possible that the McCabe's complexity is higher than the number of possible execution paths in the program?**

- **Lower than this number?**

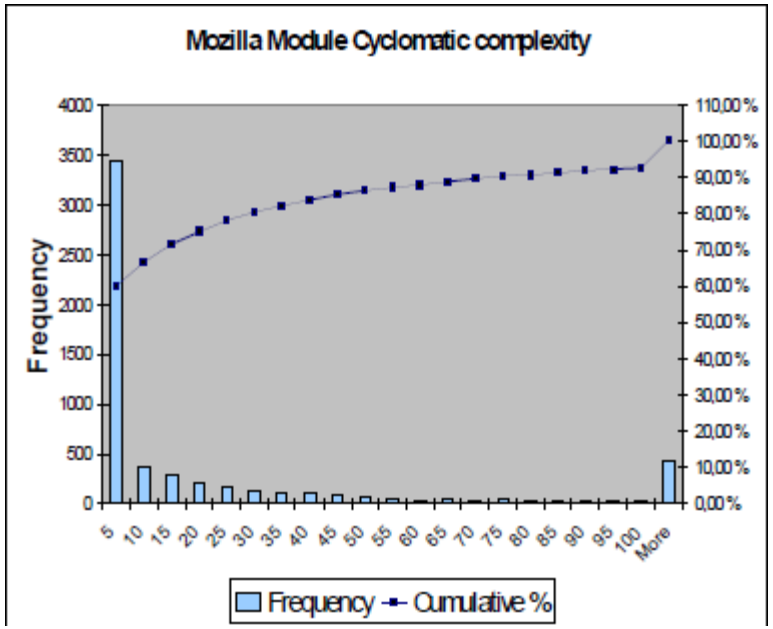
McCabe's complexity in Linux kernel

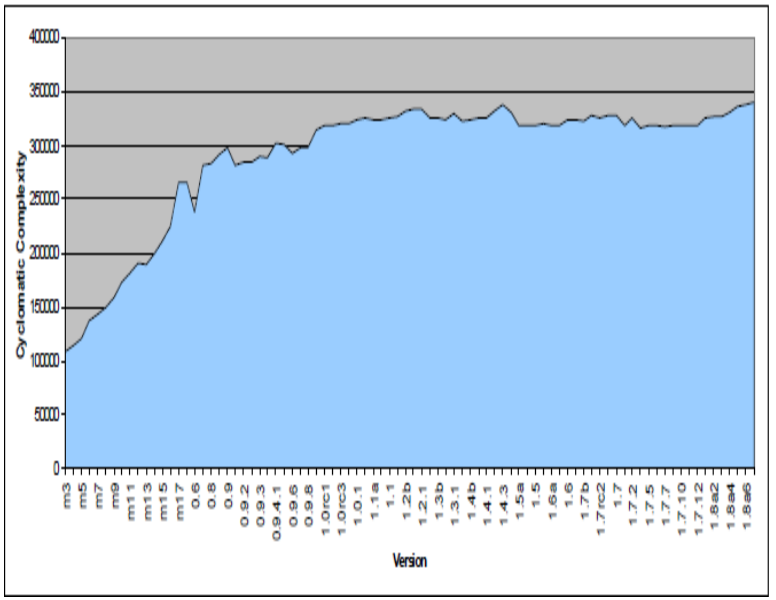


A. Israeli, D.G. Feitelson 2010

- Linux kernel
- Multiple versions and variants
  - Production (blue dashed)
  - Development (red)
  - Current 2.6 (green)

McCabe's complexity in Mozilla [Røsdal 2005]





- **Most of the modules have low cyclomatic complexity**
- **Complexity of the system seems to stabilize**

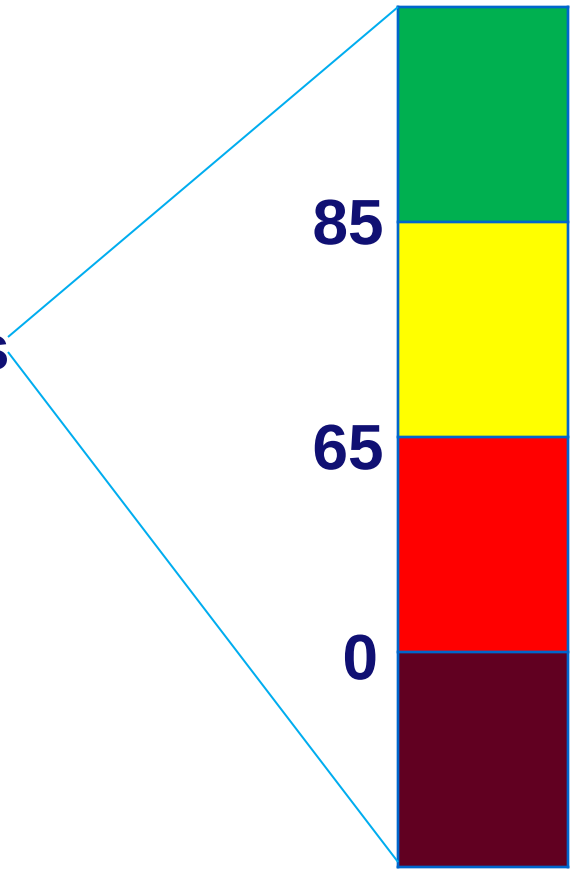
Summarizing: Maintainability index (MI) [Coleman, Oman 1994]

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

Halstead McCabe LOC

$$MI_2 = MI_1 + 50\sin\sqrt{2.46\,perCM}$$

% comments



- MI_2 can be used only if comments are meaningful
- If more than one module is considered – use average values for each one of the parameters
- Parameters were estimated by fitting to expert evaluation
  - BUT: few middle-sized systems!

McCabe complexity: Example

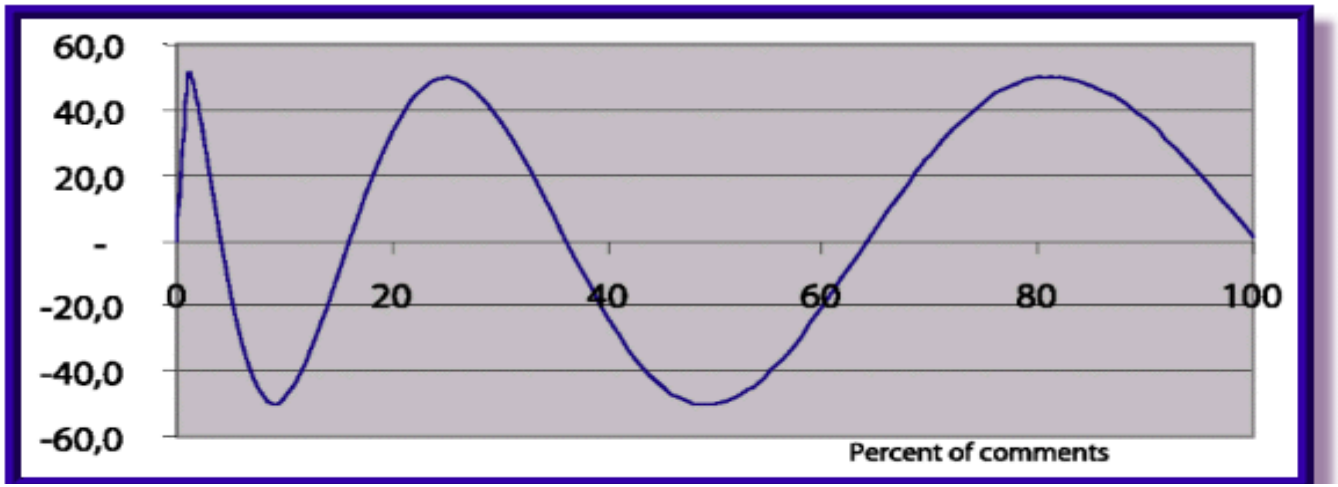
```
void sort ( int *a, int n ) {
int i, j, t;

   if ( n < 2 ) return;
   for ( i=0 ; i < n-1; i++ )  {
         for ( j=i+1 ; j < n ; j++ ) {
               if ( a[i] > a[j] ) {
                     t = a[i];
                     a[i] = a[j];
                     a[j] = t;
               }
         }
   }
}
```

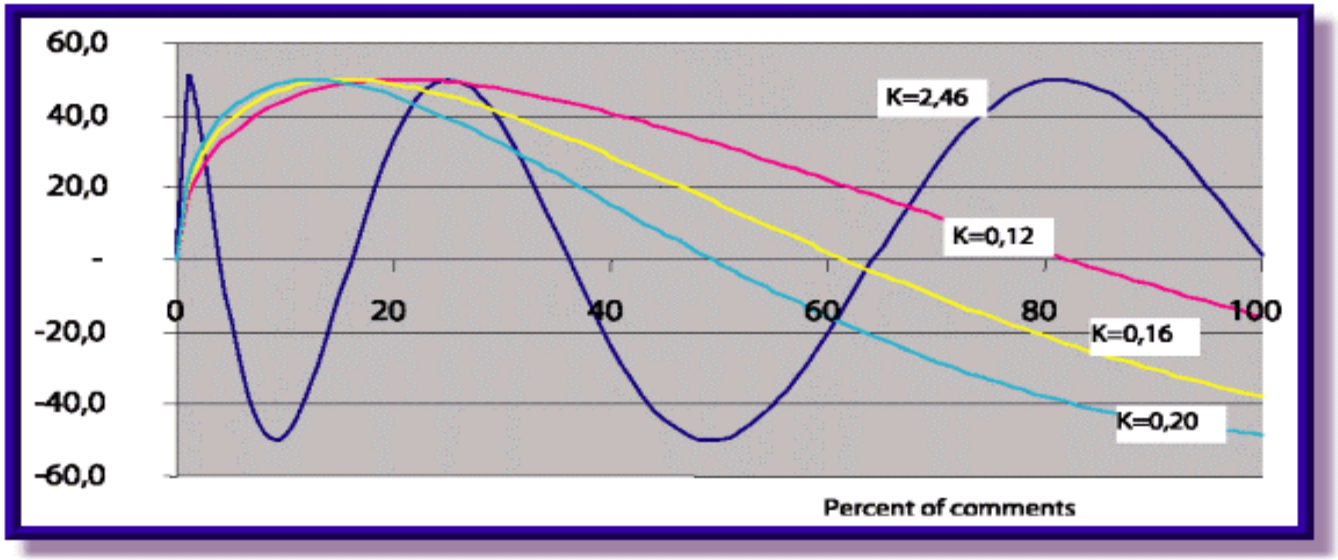
- Halstead's V ≈ 392
- McCabe's v(G) = 5
- LOC = 14
- $MI_1 \approx 96$
- Easy to maintain!

Comments?

$$50 \sin \sqrt{2.46\, perCM}$$

[Liso 2001]





Peaks:

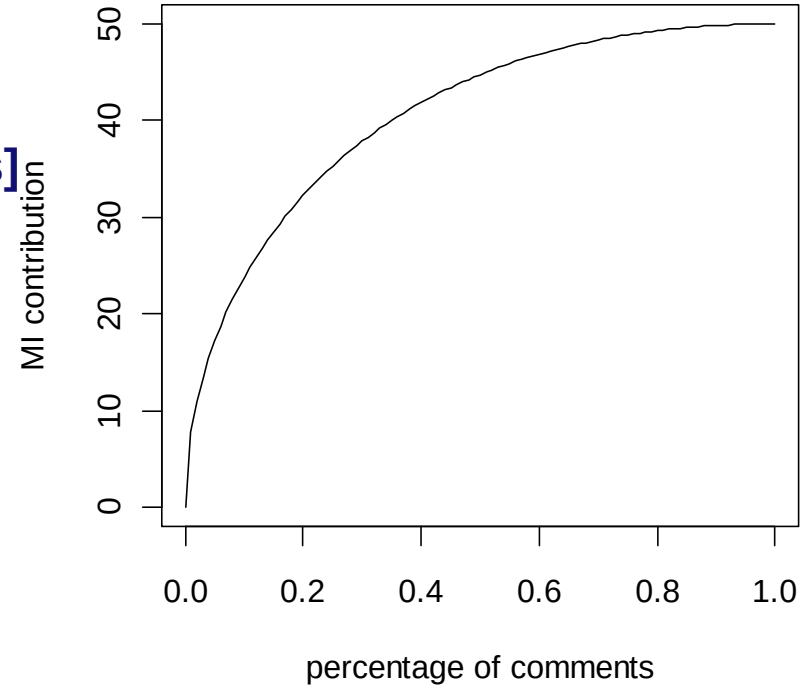
- 25% (OK),
- 1% and 81% - ???

Better:

- $0.12 \leq K \leq 0.2$

Another alternative:

- **Percentage as a fraction [0;1] – [Thomas 2008, Ph.D. thesis]**
- **The more comments – the better?**



Evolution of the maintainability index in Linux

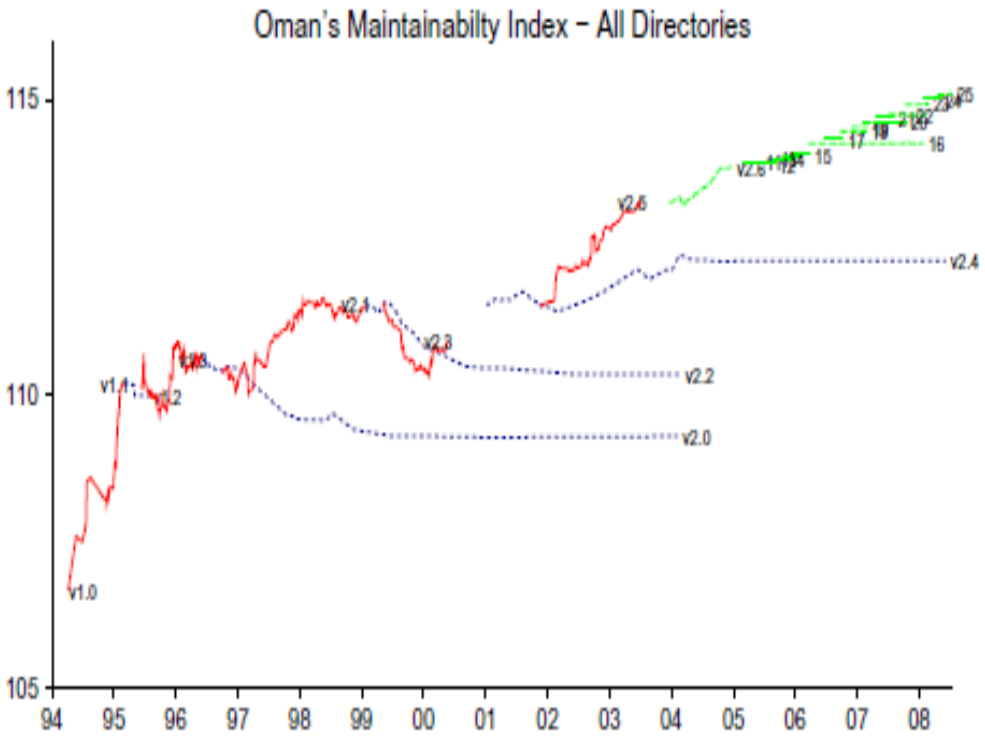


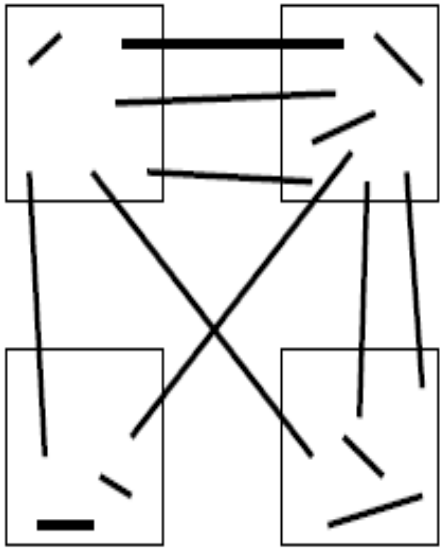
Fig. 9. Evolution of Oman's maintainability index.

- **Size, Halstead volume and McCabe complexity decrease**
- **% comments decreases as well**
  - **BUT they use the [0;1] definition, so the impact is limited**

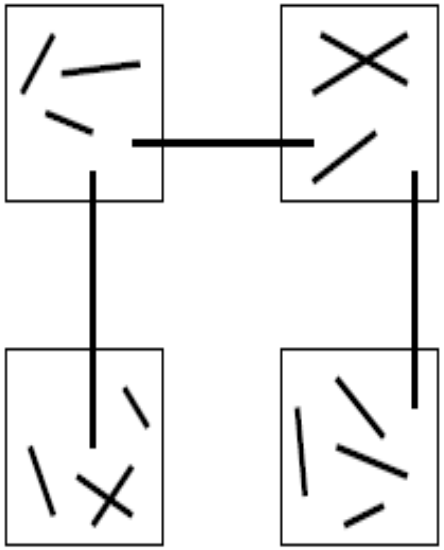
A. Israeli, D.G. Feitelson 2010

What about modularity?

Design A



Design B

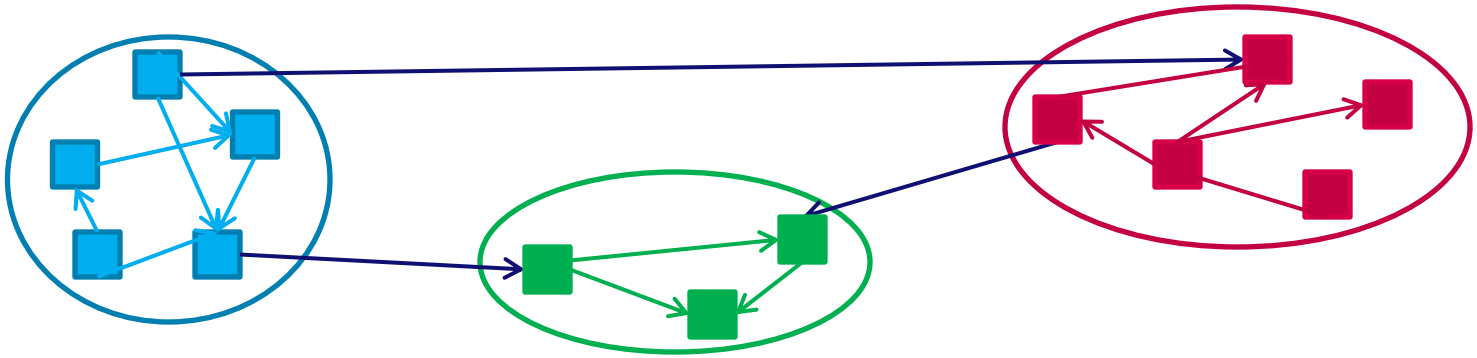


- **Cohesion: calls inside the module**
- **Coupling: calls between the modules**

	A	B
Cohesion	Lo	Hi
Coupling	Hi	Lo

- **Squares are modules, lines are calls, ends of the lines are functions.**
- **Which design is better?**

Do you remember?



- **Many intra-package dependencies: high cohesion**

$$A_i = \frac{\mu_i}{N_i^2} \quad \textbf{or} \quad A_i = \frac{\mu_i}{N_i(N_i - 1)}$$

- **Few inter-package dependencies: low coupling**

$$E_{i,j} = \frac{\varepsilon_{i,j}}{2N_iN_j}$$

- **Joint measure**

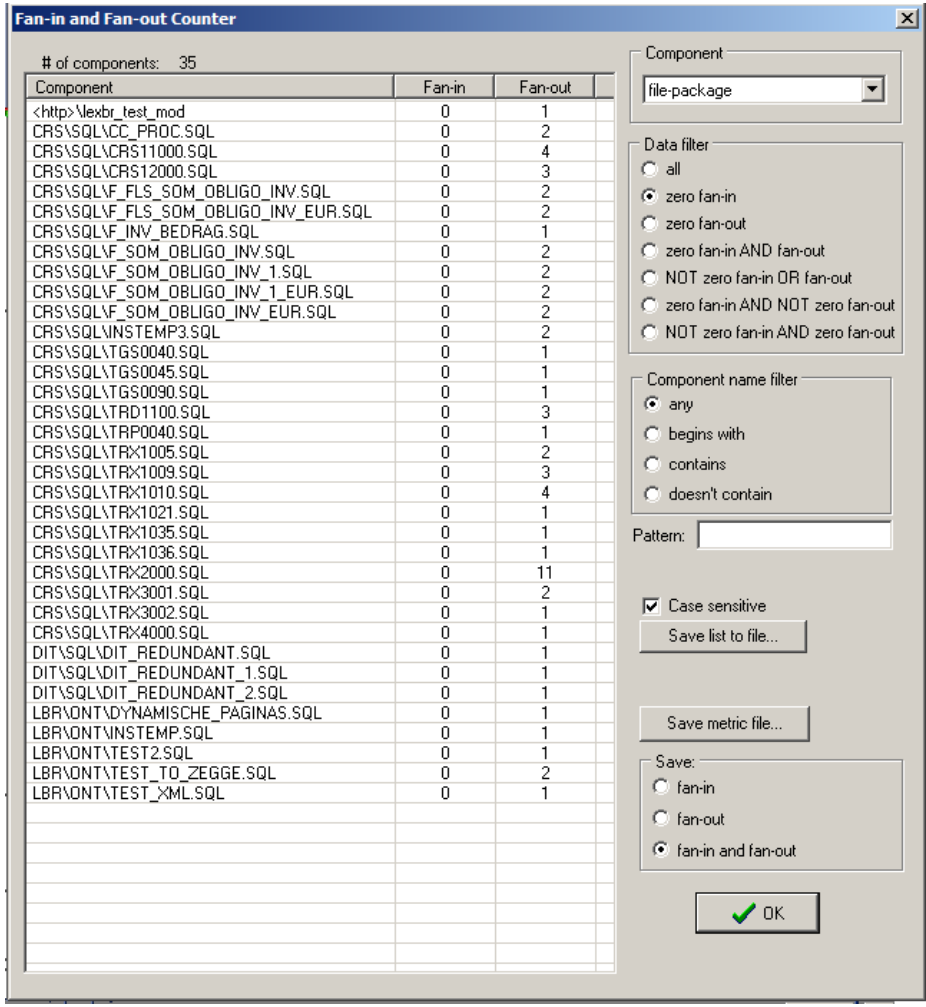
$$MQ = \frac{1}{k}\sum_{i=1}^{k} A_i - \frac{2}{k(k-1)}\sum_{i=1}^{k-1}\sum_{j=i+1}^{k} E_{i,j}$$

k **- Number of packages**

Modularity metrics: Fan-in and Fan-out

- **Fan-in of M: number of modules calling functions in M**
- **Fan-out of M: number of modules called by M**

- **Modules with fan-in = 0**
- **What are these modules?**
  - **Dead-code**
  - **Outside of the system boundaries**
  - **Approximation of the “call” relation is imprecise**



Henry and Kafura's information flow complexity [HK 1981]

- **Fan-in and fan-out can be defined for procedures**
  - **HK: take global data structures into account:**
    - **read for fan-in,**
    - **write for fan-out**

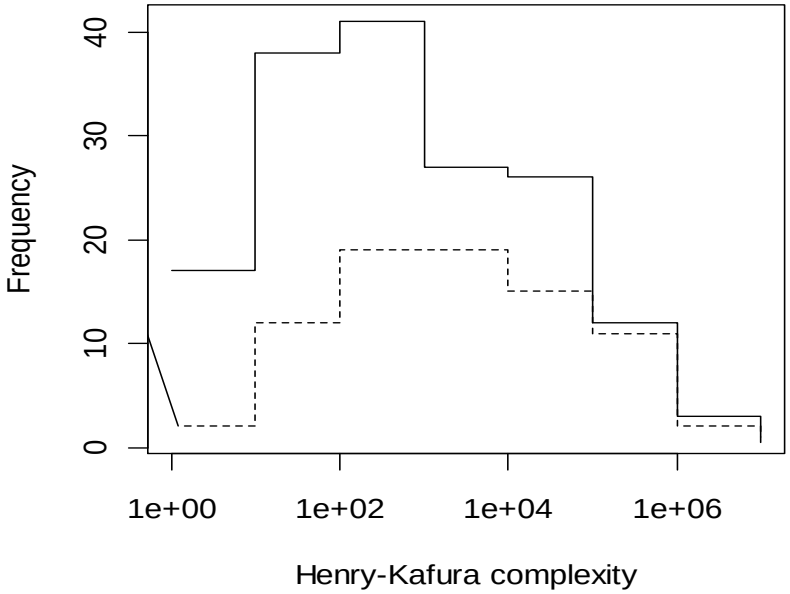
- **Henry and Kafura: procedure as HW component connecting inputs to outputs**

$$hk = sloc * (fanin * fanout)^2$$

- **Shepperd**

$$s = (fanin * fanout)^2$$

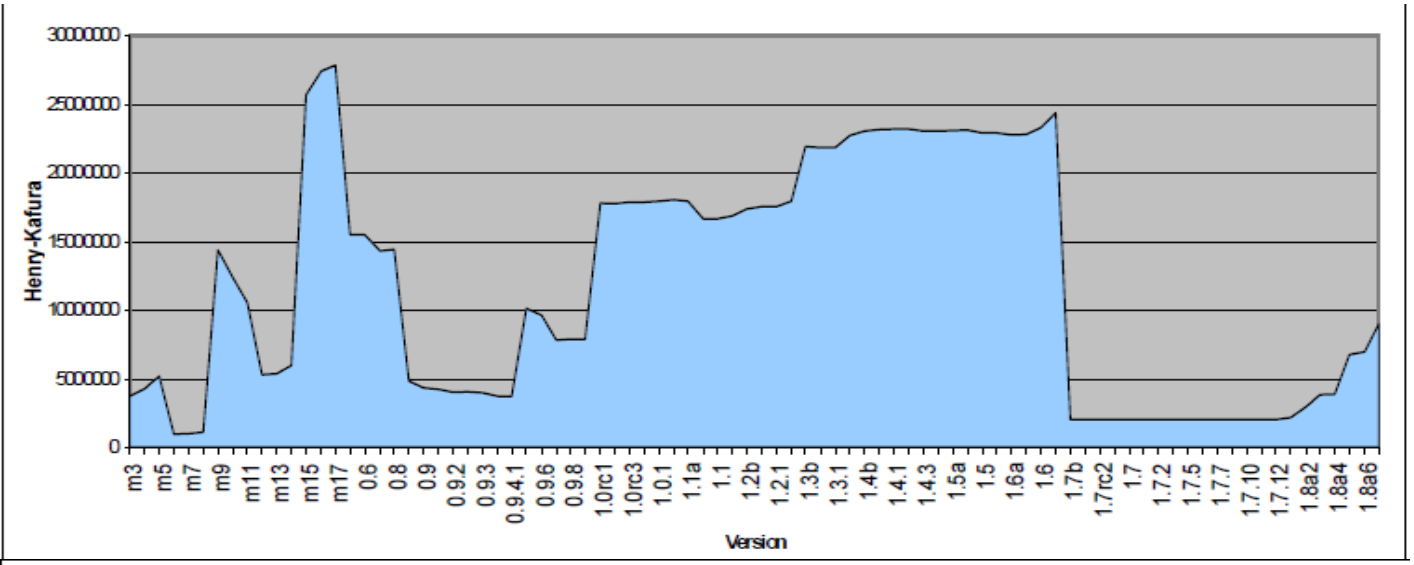
Information flow complexity of Unix procedures

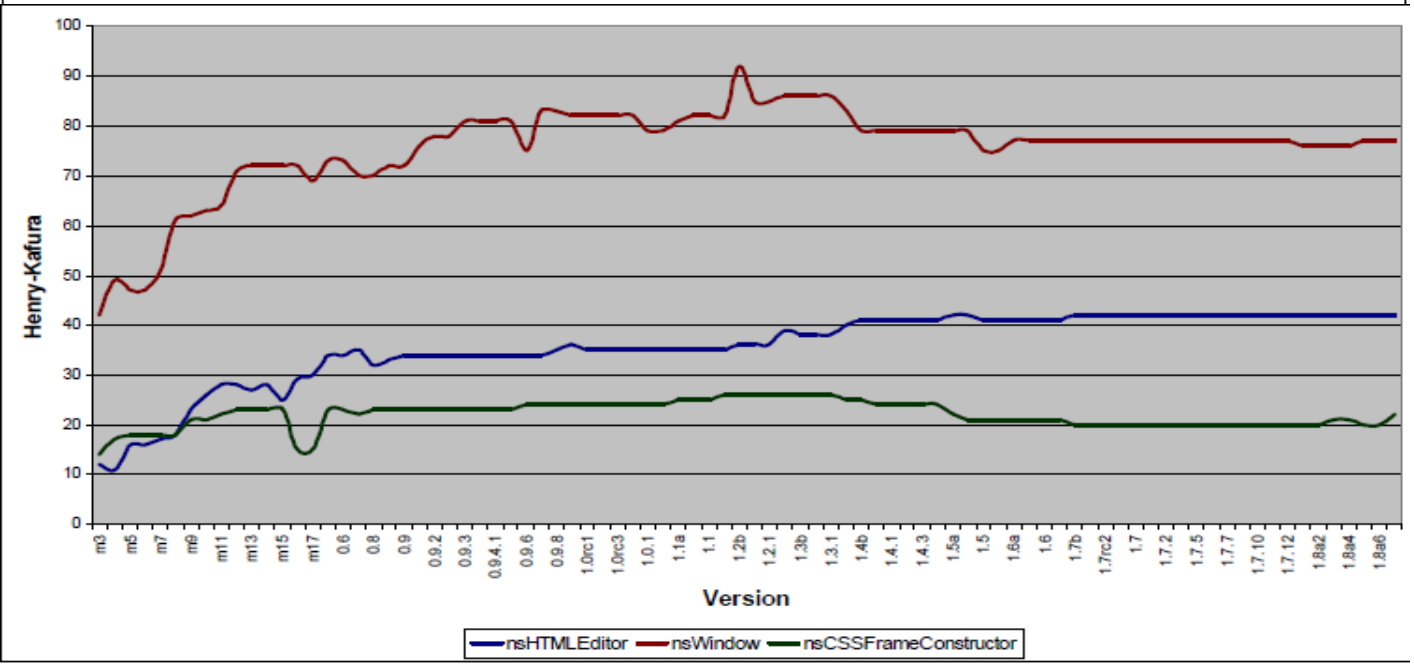


Module Name	Complexity	Procedure Complexity of Three Largest Procedures	Percent
buf	3541083	3468024	98
file	33062	29425	89
filesys	268807	254080	95
inode	13462921	12984995	96
kl11	3262	2120	65
lp11	855	829	97
mount	135503	135084	99
proc	436151	379693	87
text	24886	24831	99

- **Solid – #procedures within the complexity range**
- **Dashed - #changed procedures within the complexity range**
- **Highly complex procedures are difficult to change but they are changed often!**
- **Complexity comes from the three largest procedures**

Evolution of the information flow complexity

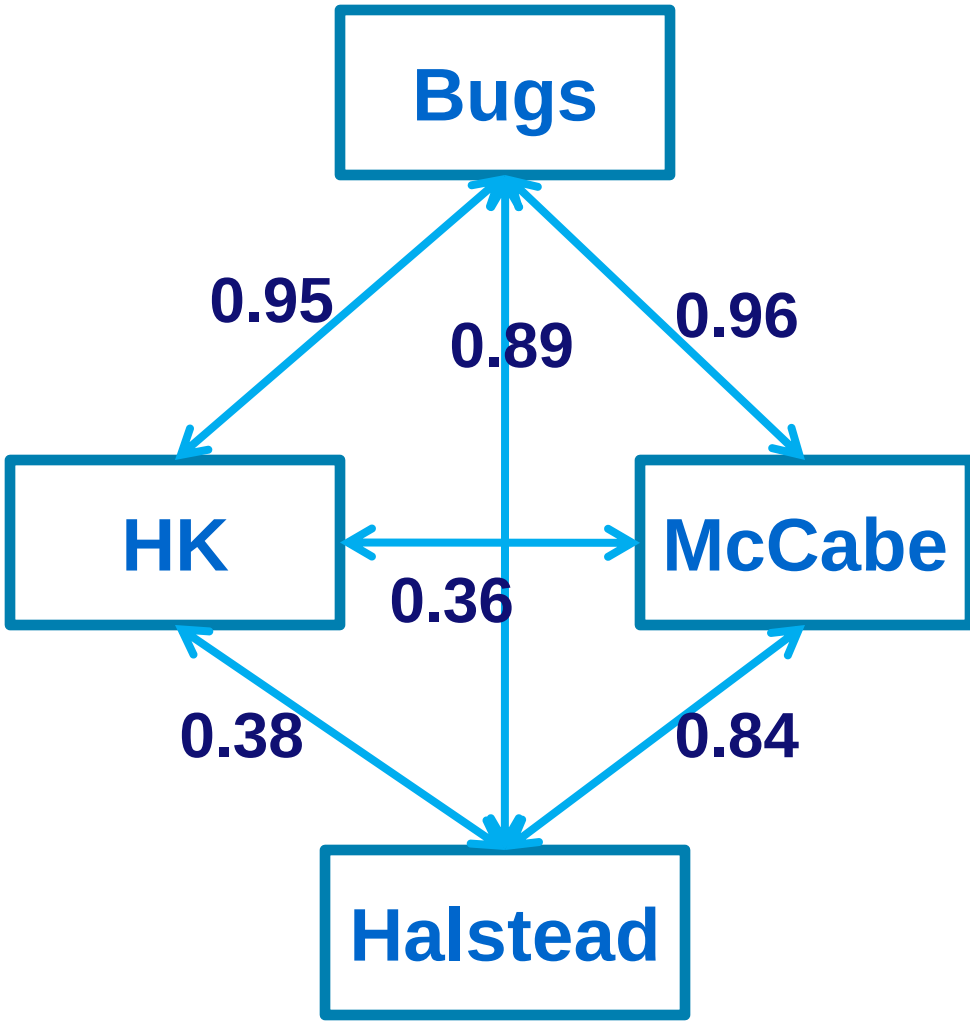




- Mozilla
- Shepperd version
- Above: Σ the metrics over all modules
- Below: 3 largest modules
- What does this tell?

Summary so far…

- Complexity metrics
  - Halstead's effort
  - McCabe (cyclomatic)
  - Henry Kafura/Shepperd (information flow)
- Are these related?
- And what about bugs?
- Harry,Kafura,Harris 1981
  - 165 Unix procedures
- What does this tell us?



From imperative to OO

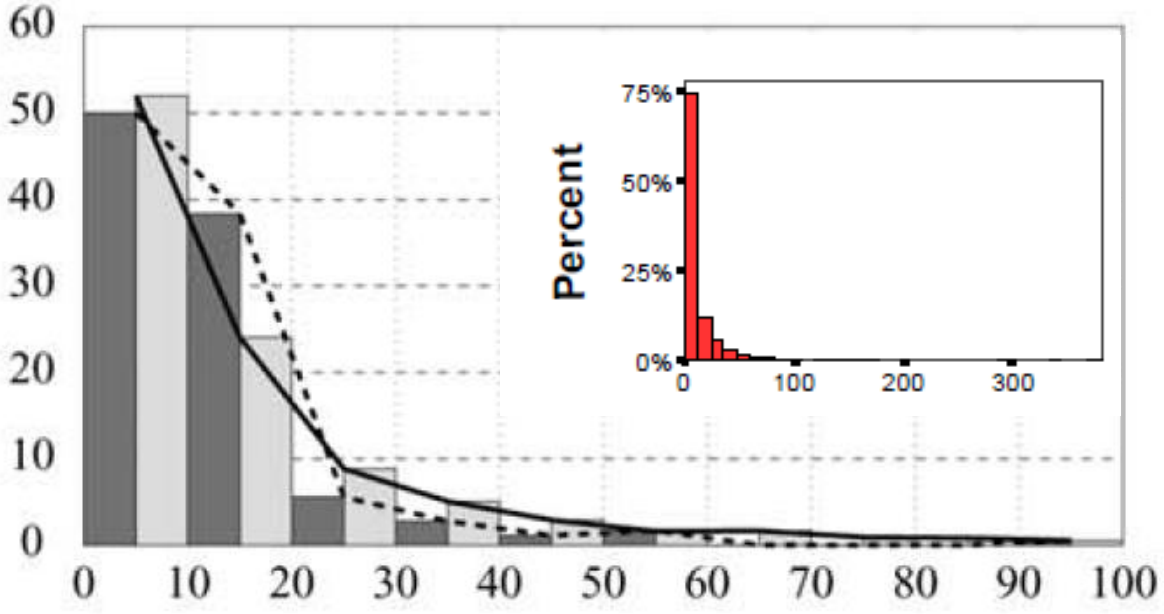
- All metrics so far were designed for imperative languages
  - Applicable for OO
    - On the method level
    - Also
      - Number of files → number of classes/packages
      - Fan-in → afferent coupling (C_a)
      - Fan-out → efferent coupling (C_e)
  - But do not reflect OO-specific complexity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

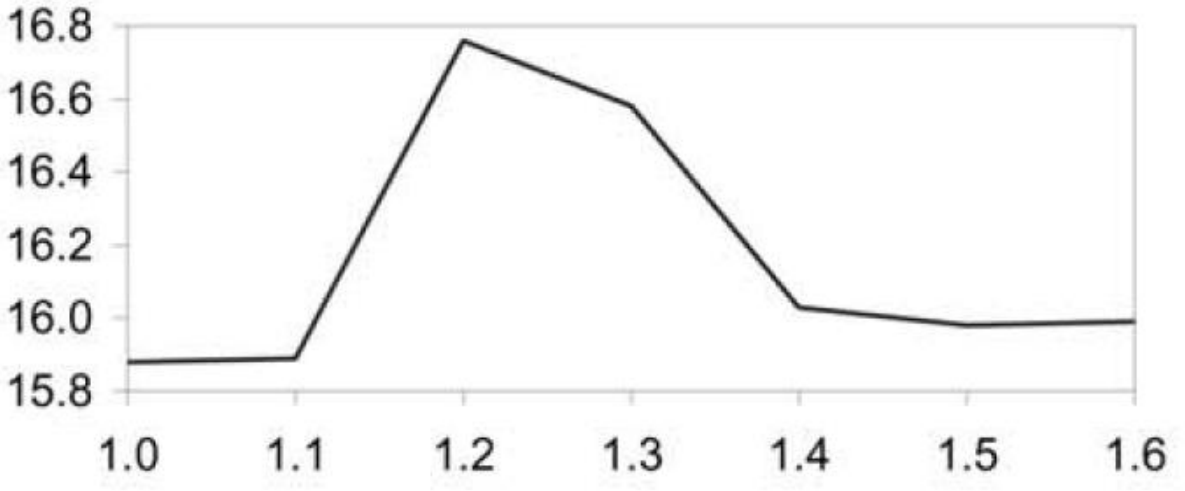
- **WMC – weighted methods per class**
  - **Sum of metrics(m) for all methods m in class C**
  - **Popular metrics: McCabe's complexity and unity**
  - **WMC/unity = number of methods**
  - **Statistically significant correlation with the number of defects**



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

Chidamber and Kemerer

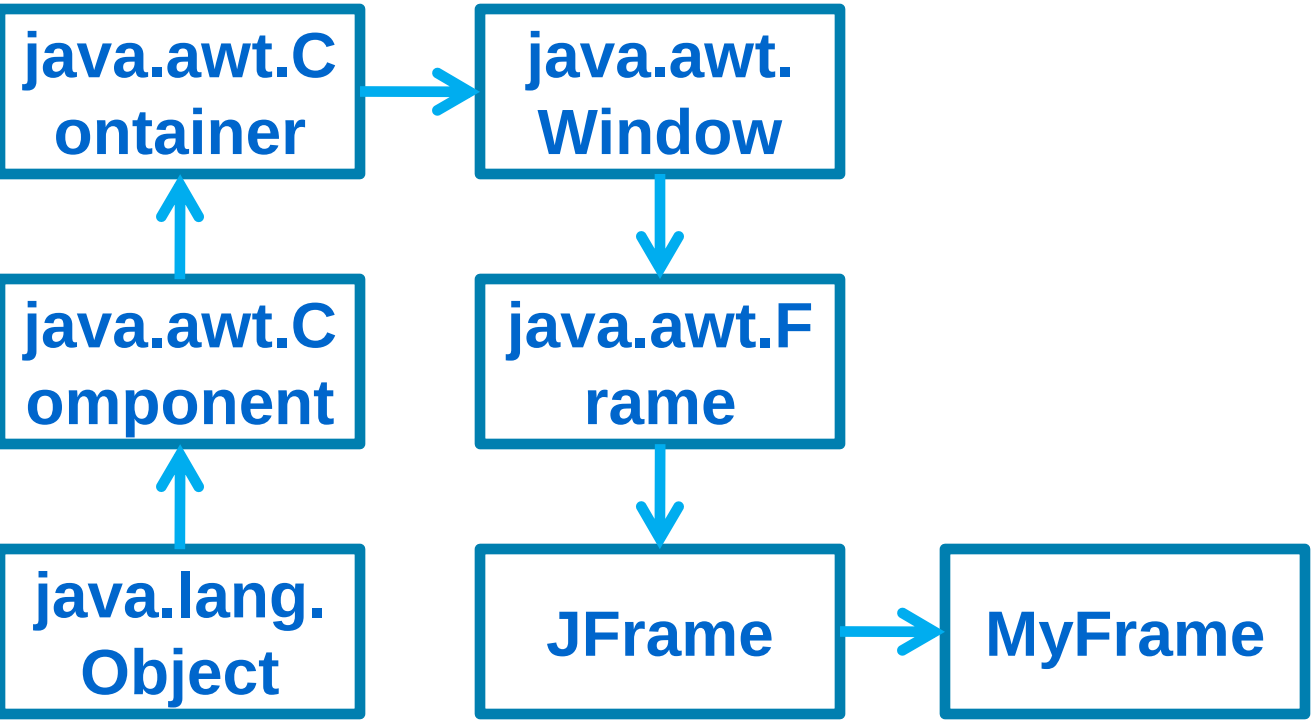
- **WMC – weighted methods per class**
  - **Sum of metrics(m) for all methods m in class C**
  - **Popular metrics: McCabe's complexity and unity**
  - **WMC/unity = number of methods**
  - **Statistically significant correlation with the number of defects**



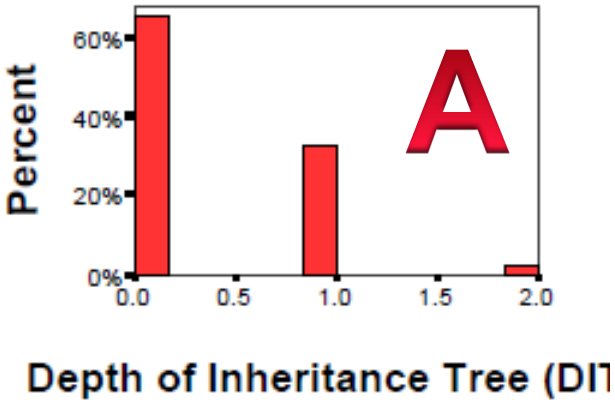
- **WMC/unity**
- **Gyimothy et al.**
- **Average**

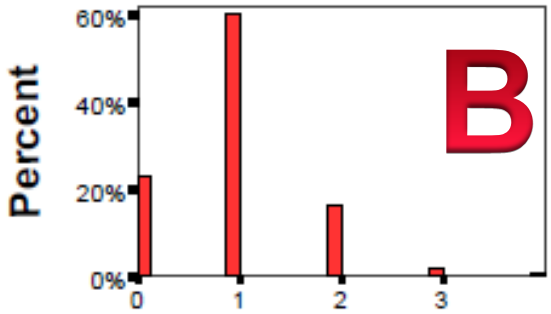
Depth of inheritance - DIT

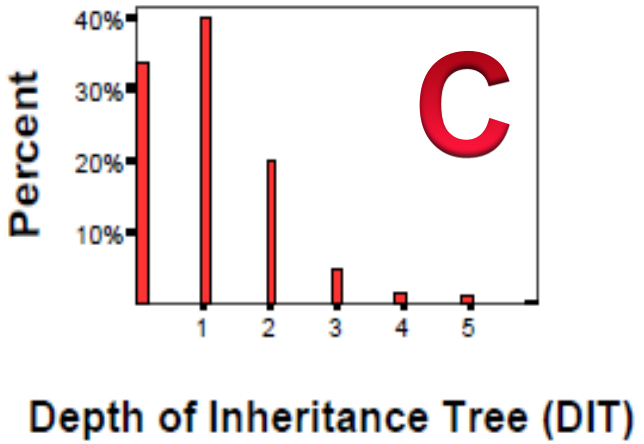
- **Variants: Were to start and what classes to include?**
  - **1, JFrame is a library class, excluded**
  - **2, JFrame is a library class, included**
  - **7**



DIT – what is good and what is bad?

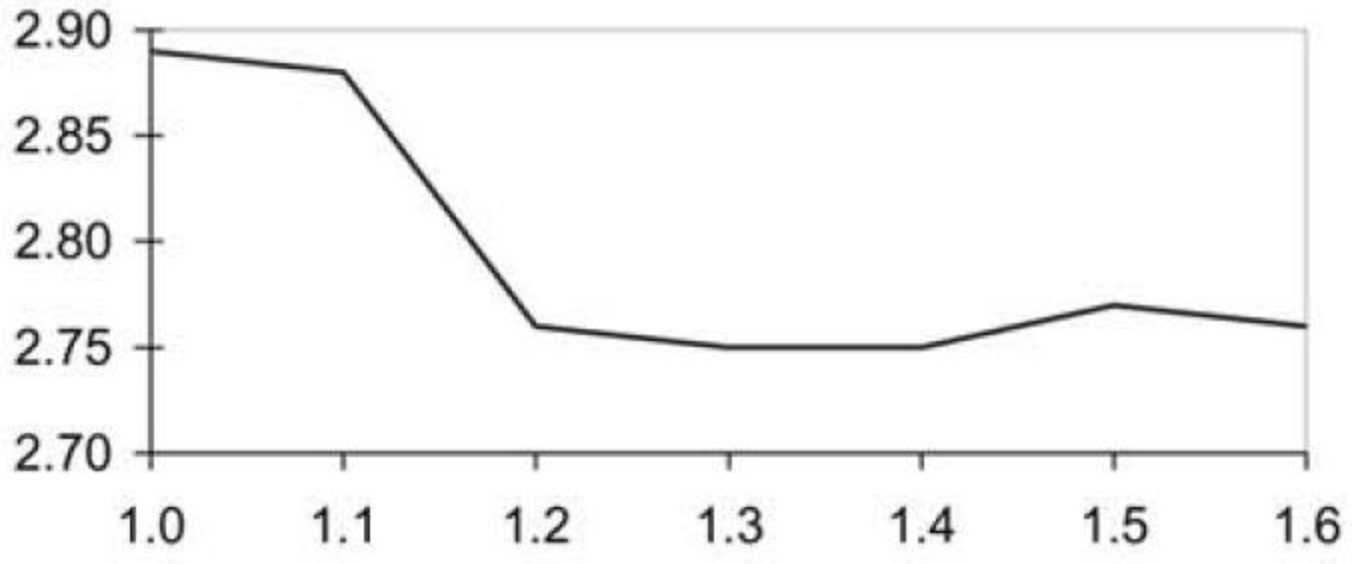






- Three NASA systems
- What can you say about the use of inheritance in systems A, B and C?
- Observation: quality assessment depends not just on one class but on the entire distribution

Average DIT in Mozilla



- **How can you explain the decreasing trend?**

Other CK metrics

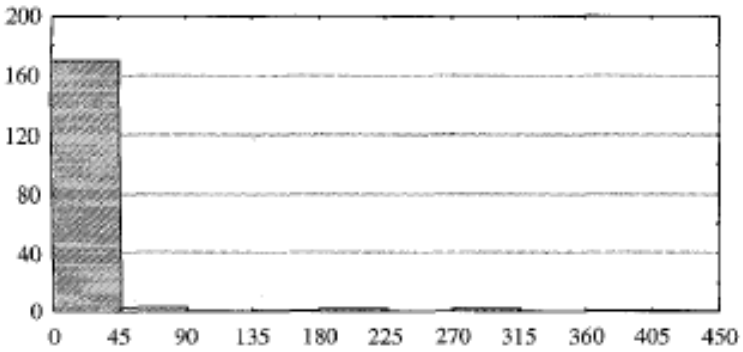
- NOC – number of children
- CBO – coupling between object classes
- RFC - #methods that can be executed in response to a message being received by an object of that class.
- More or less “exponentially” distributed

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**



[BBM] 180 classes

- **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**

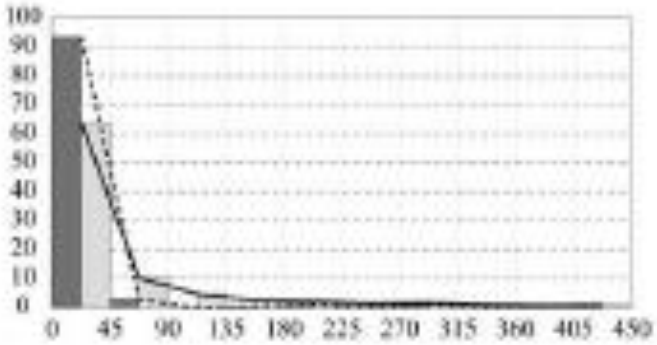
Discriminative ability is insufficient

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

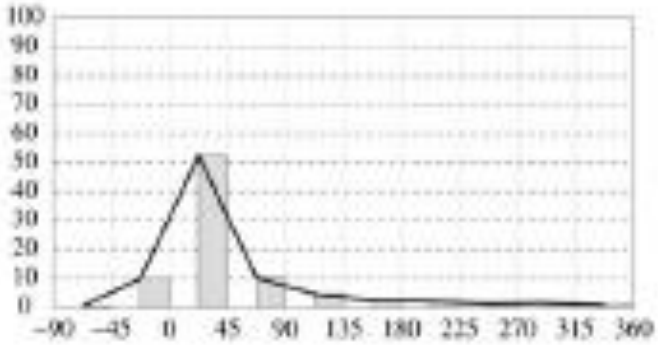
First solution: LCOMN

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

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

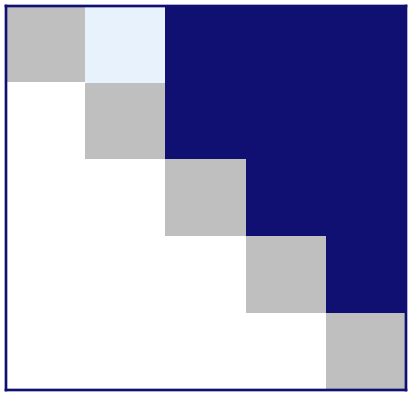


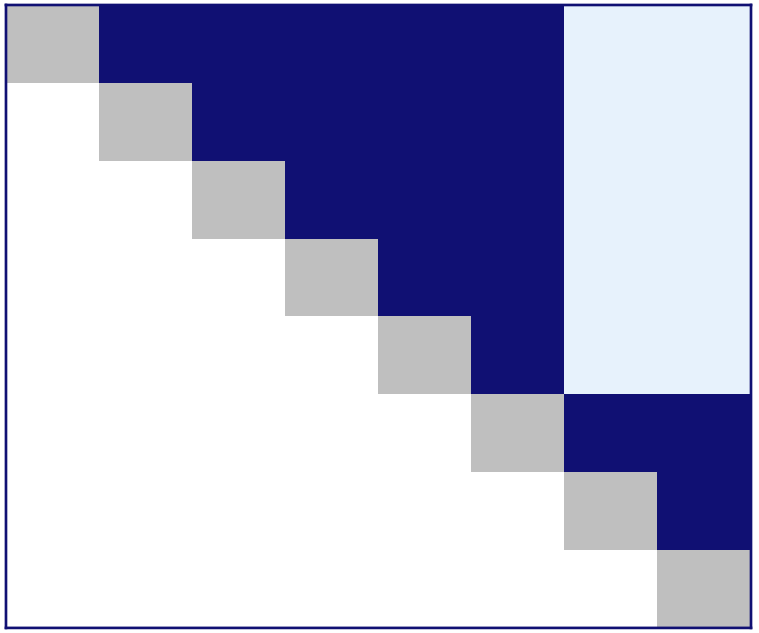
LCOM



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**

$$m(V_i) = m \text{ and } LCOM = 0$$

- **No cohesion: every method accesses a unique variable**

$$m(V_i) = 1 \text{ and } LCOM = 1$$

- **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

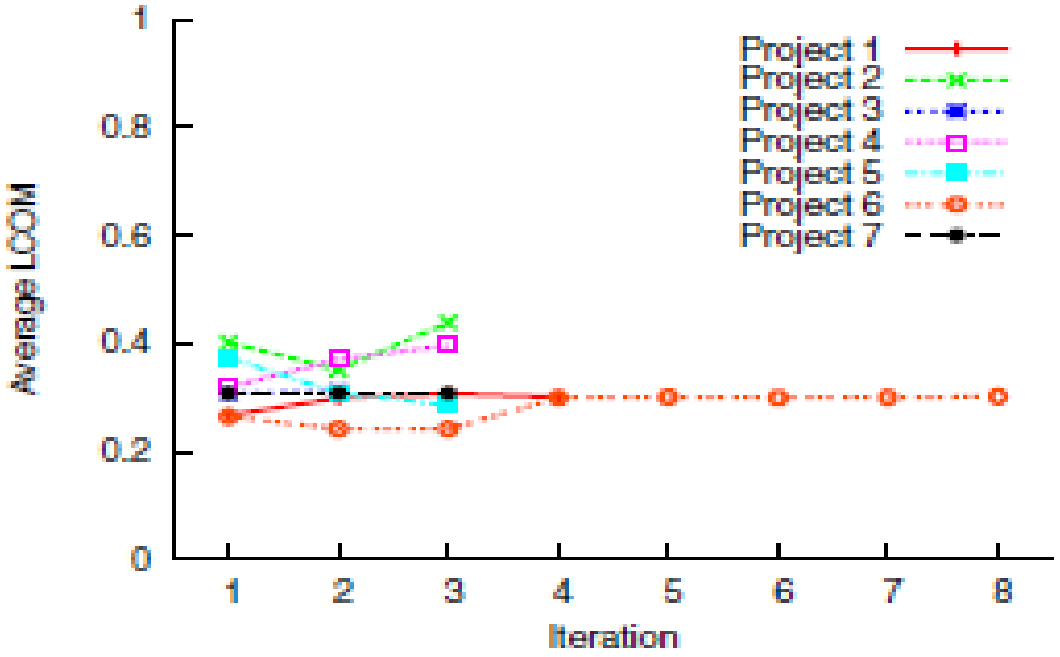
$$\frac{\left(\frac{1}{v}\sum_{i=1}^{v} m(V_i)\right) - m}{1-m} = \frac{-m}{1-m} = 1 + \frac{1}{m-1}$$

Hence

LCOM is undefined for m = 1

LCOM ≤ 2

Evolution of LCOM [Henderson-Sellers et al.]

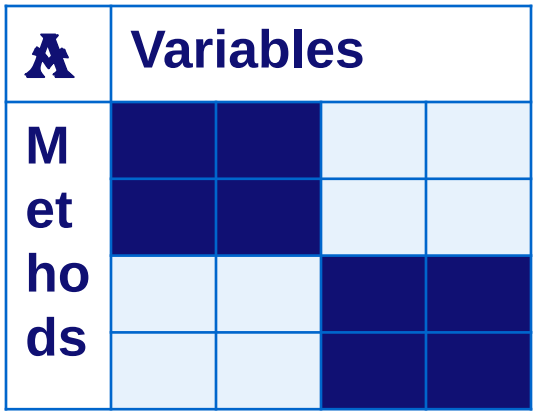


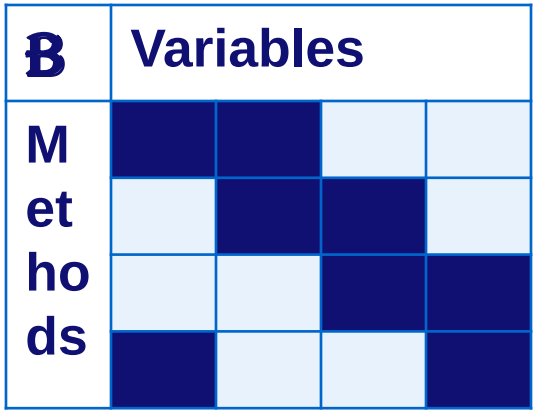
Sato, Goldman, Kon 2007

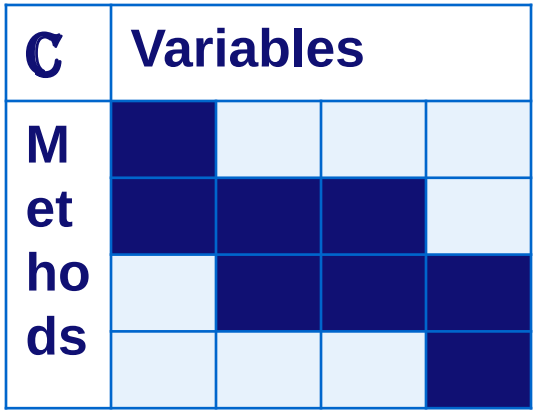
- **Project 6 (commercial human resource system) suggests stabilization, but no similar conclusion can be made for other projects**

Shortcomings of LCOM [Henderson-Sellers]

Due to [Fernández, Peña 2006]





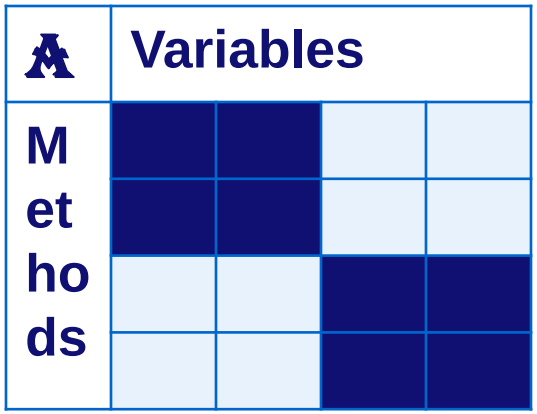


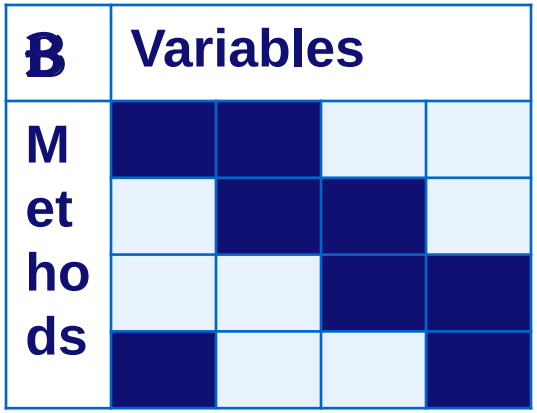
- **Method-variable diagrams: dark spot = access**
- **LCOM(A) ? LCOM(B) ?LCOM(C) ?**

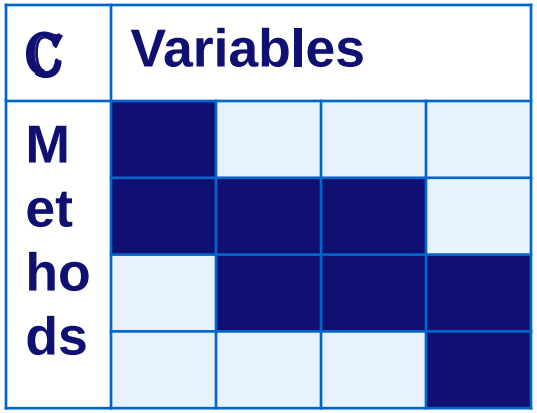
$$\frac{\left(\frac{1}{v}\sum_{i=1}^{v} m(V_i)\right) - m}{1 - m}$$

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]

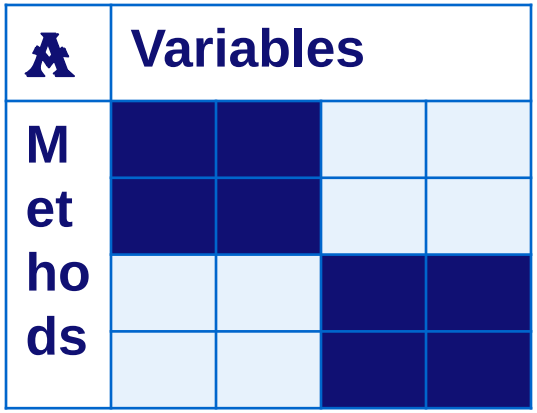
- LCOM as the number of strongly connected components in the following graph
  - Vertices: methods
    - except for getters/setters
  - Edge between a and b, if
    - a and b access the same variable

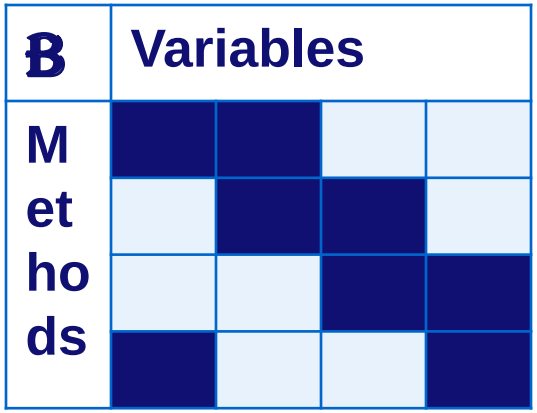
- LCOM values
  - 0, no methods
  - 1, cohesive component
  - 2 or more, lack of cohesion

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)

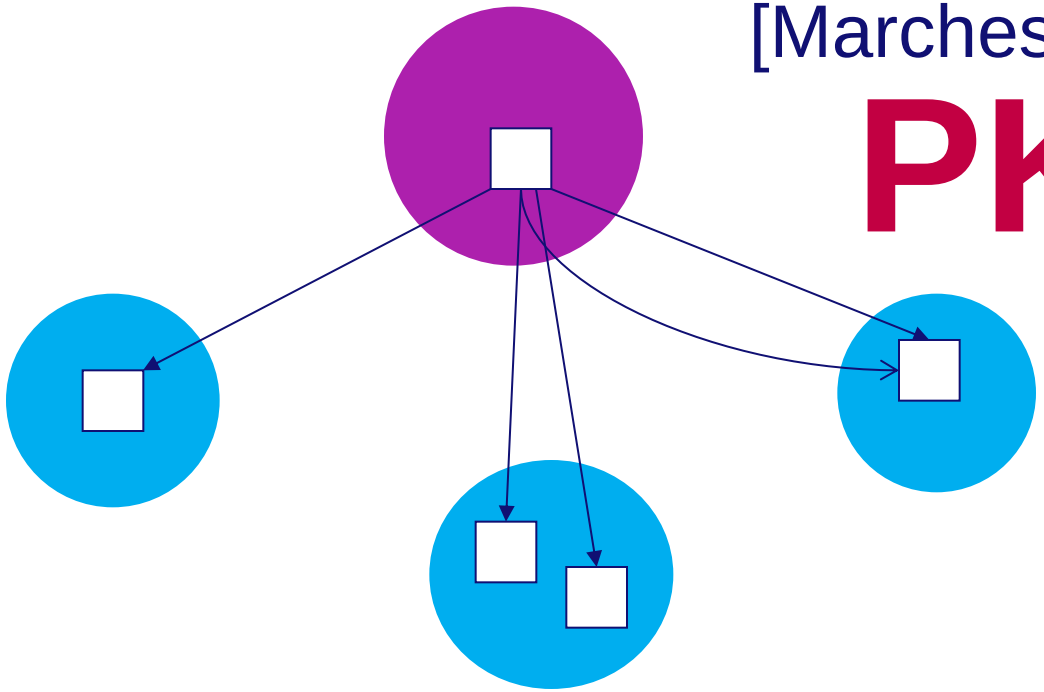
Metrics so far…

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**
- **Aggregations of class metrics**

“Fan-out”



[Marchesi 1998] [Martin 2000]

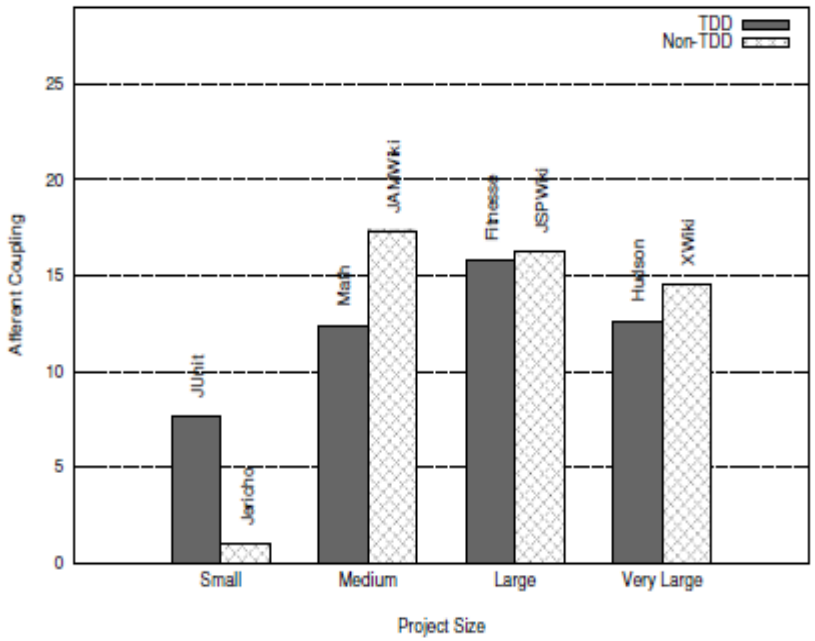
PK_1 or R: 5

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)



- 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)

[Hilton 2009]

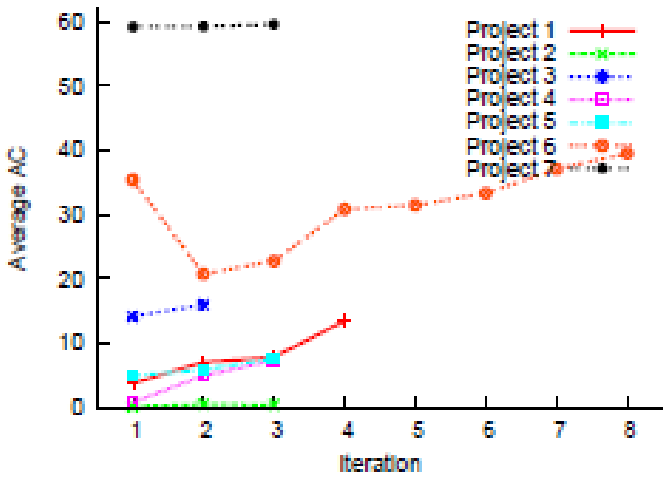
More fan-in and fan-out

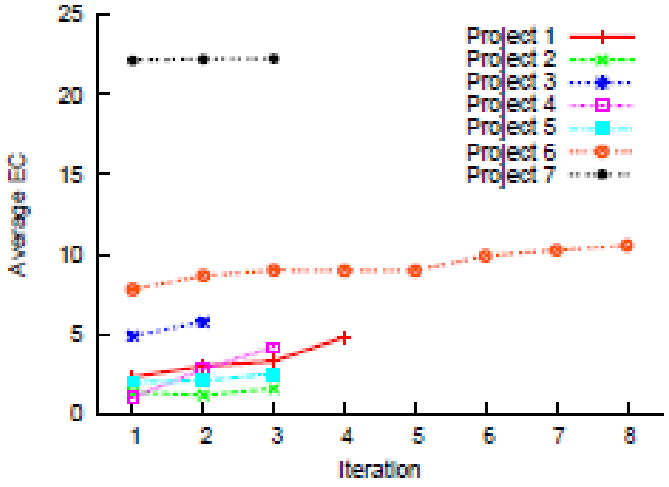
- "Fan-in" similarly to the "Fan-out"
  - Afferent coupling (Martin)
  - PK_2 (Marchesi)

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

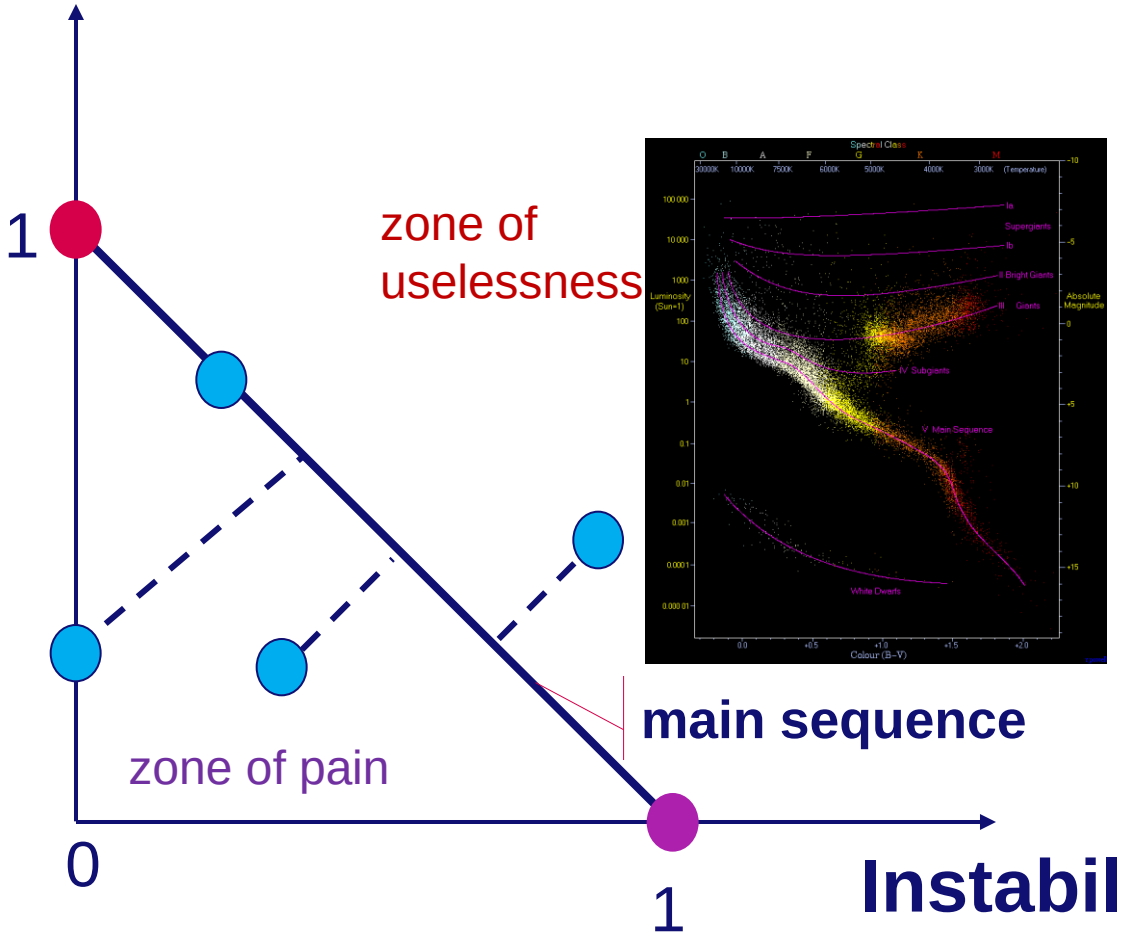
It must be **generic** enough to be adaptable to evolving requirements, either without or with only minimal modifications

Hence: contradictory criteria

D_n – Distance from the main sequence

Abstractness =

#AbstrClasses/#Classes



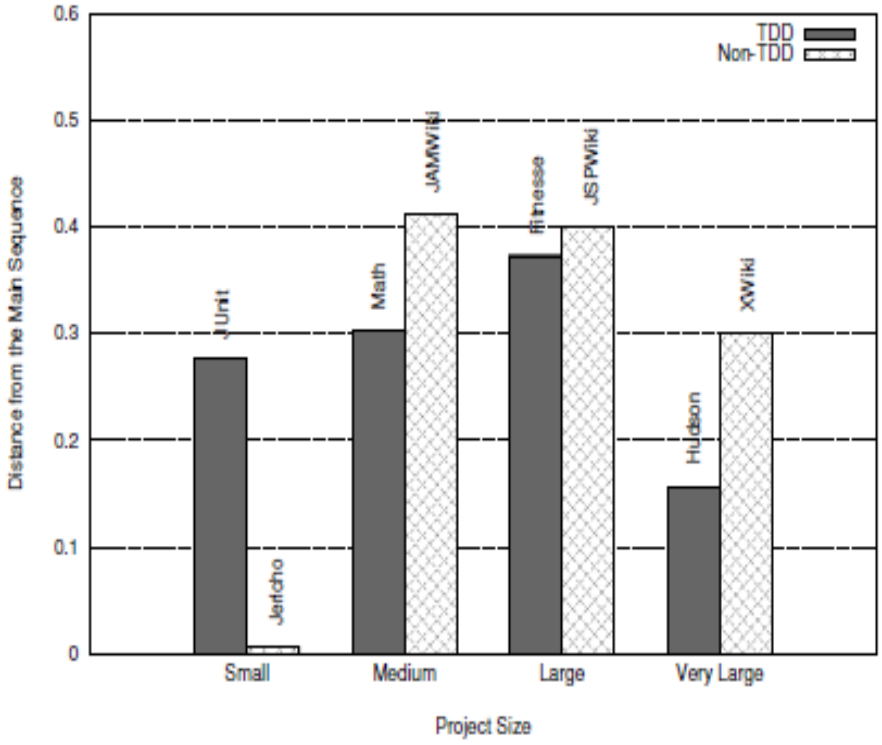
D_n =

| Abstractness +

Instability – 1 |

[R.Martin 1994]

Normalized distance from the main sequence



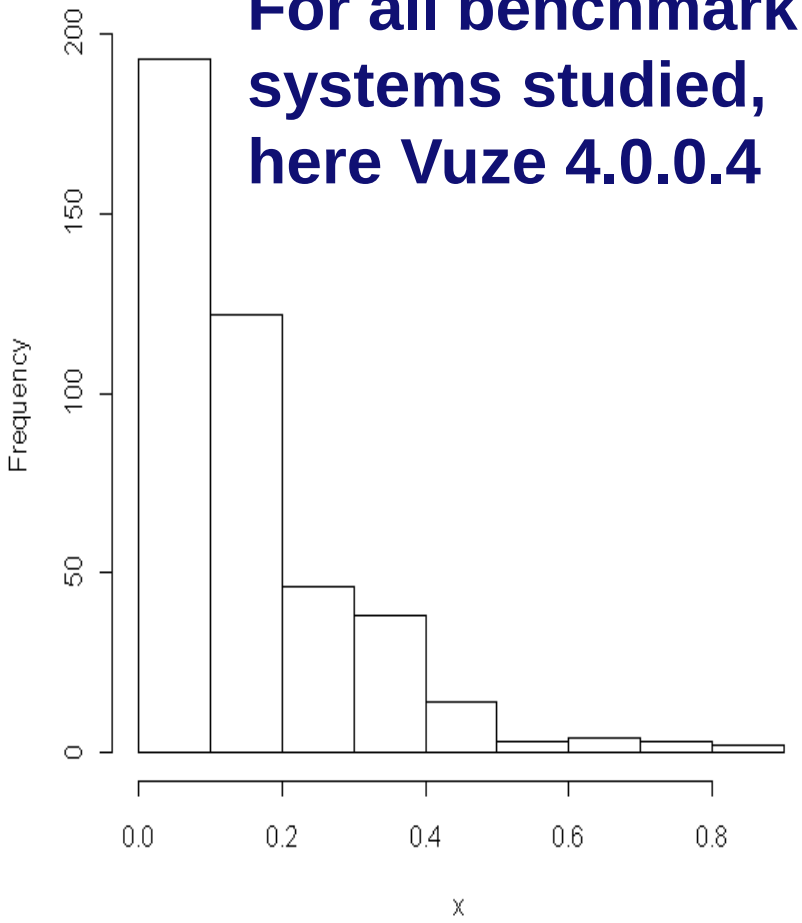
- **Dark: TDD, light: no-TDD**
- **Test-driven development positively affects D_n**
  - **The lower D_n - the better.**
- **The same exception (Jericho vs. JUnit)**

[Hilton 2009]

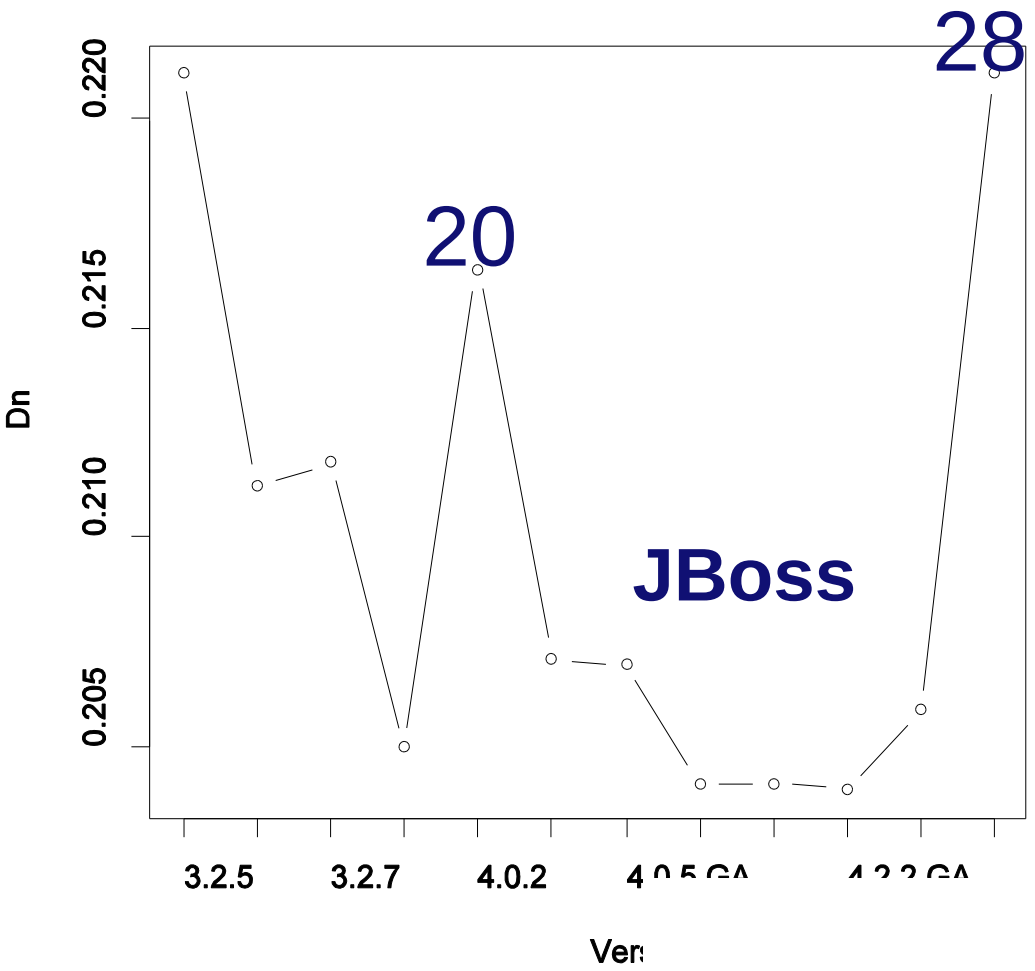
Distribution and evolution

Exponential distribution

For all benchmark systems studied, here Vuze 4.0.0.4



Peak: many feature requests (average Dn)



PASTA [Hautus 2002]

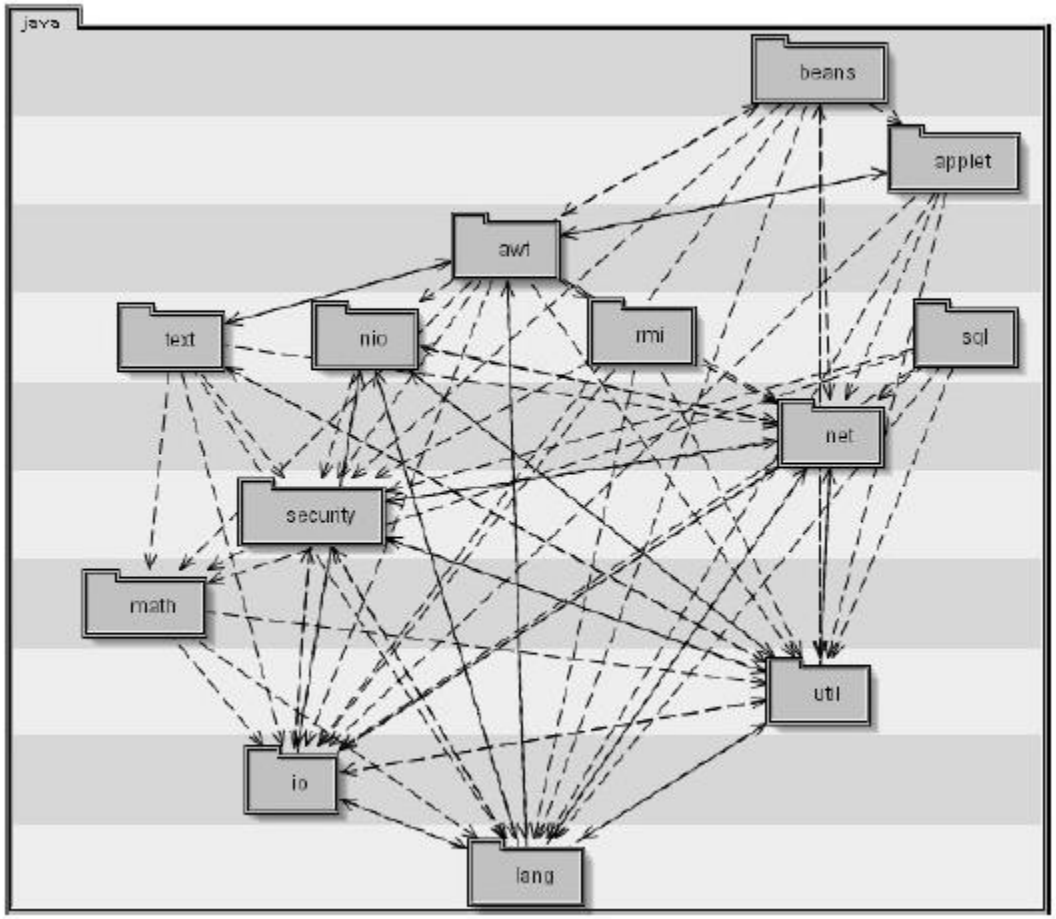
- PASTA – Package structure analysis tool

- 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

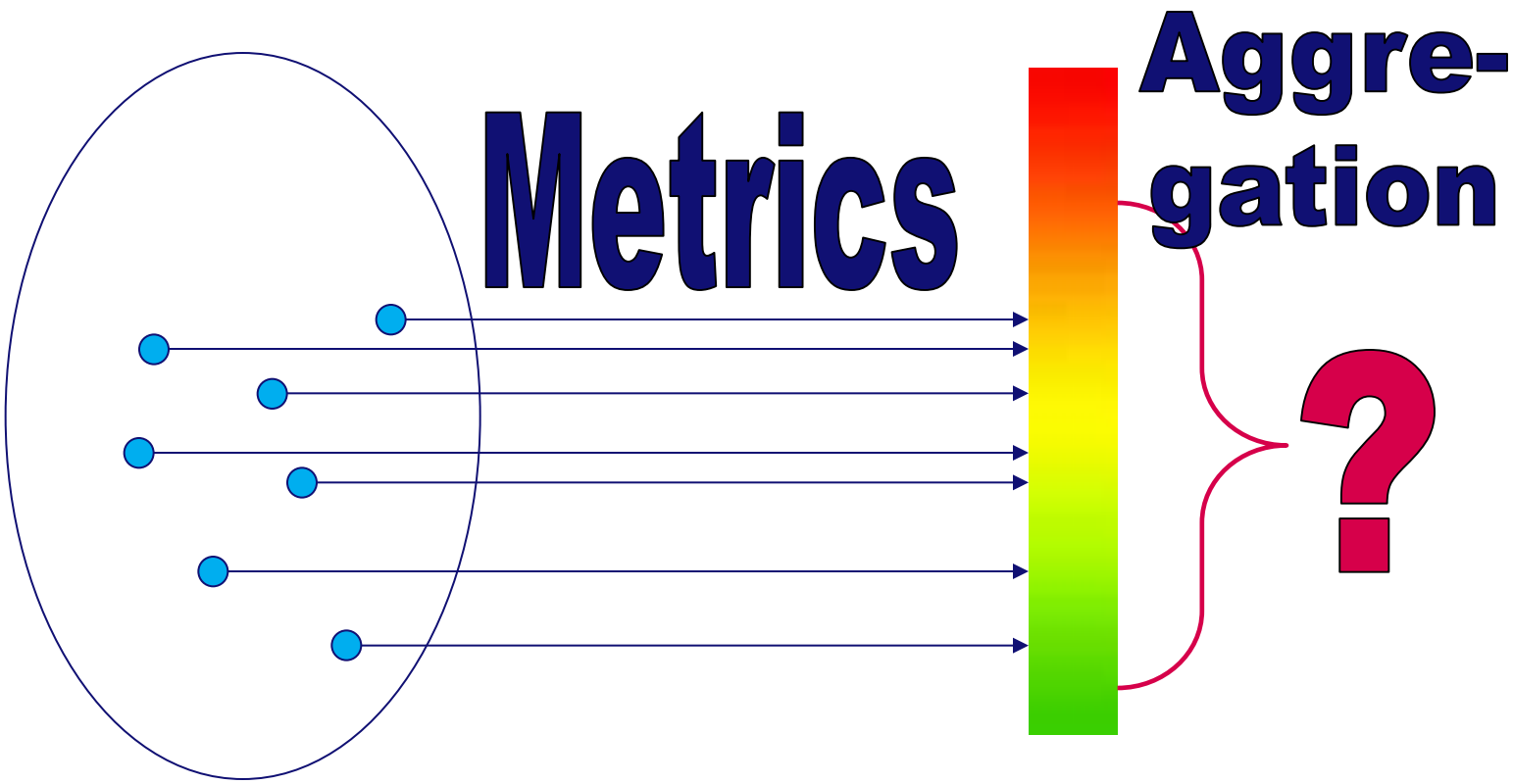
Related to NOC and NOM

Related to nesting, strongly connected components, abstractness and PK_2

Summary so far: 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**
- **Next: aggregations of class metrics**

Metrics for higher-level objects as aggregation of metrics for low-level objects



Aggregation techniques

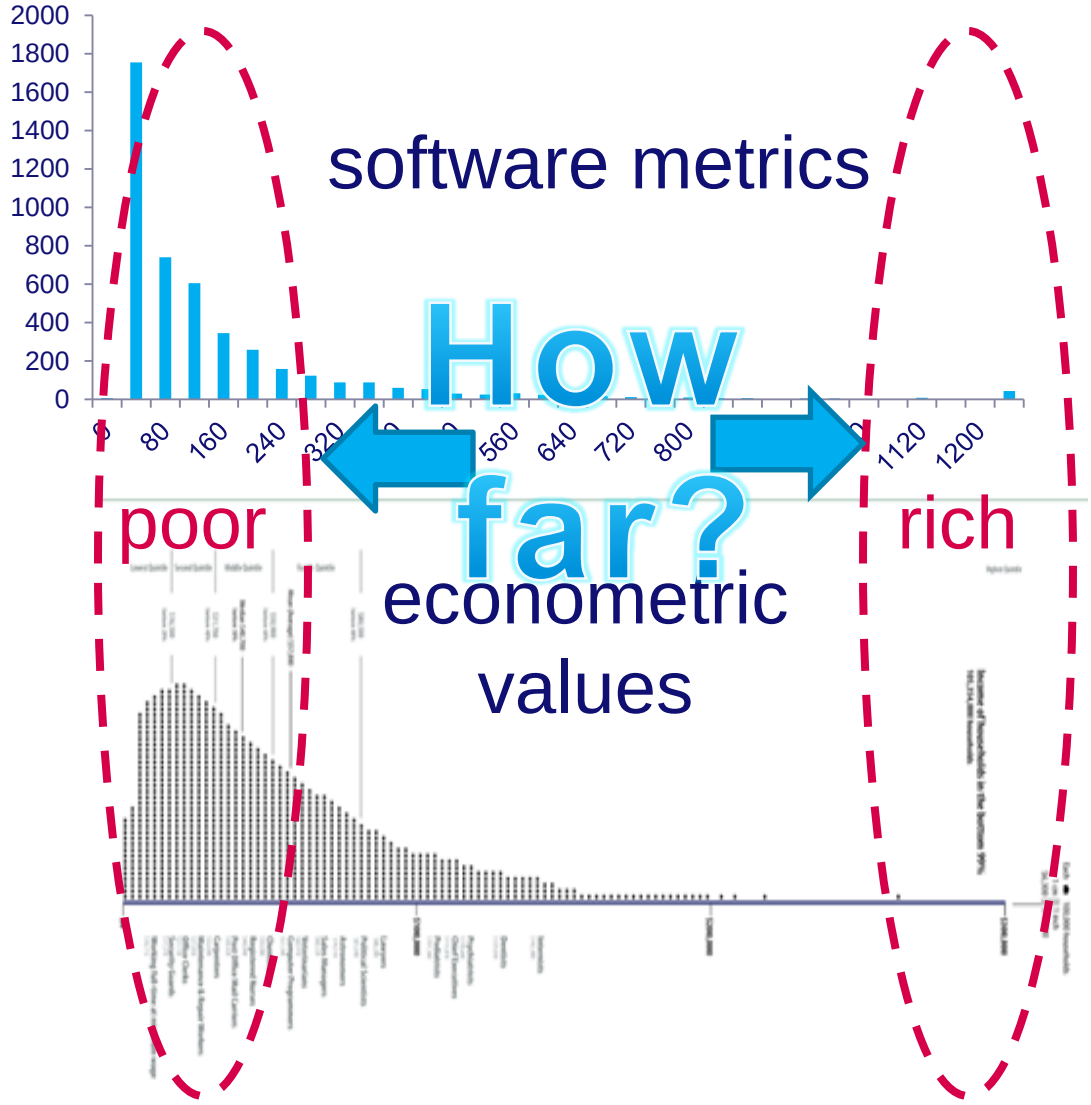
- **Metrics-independent**
  - **Applicable for any metrics to be aggregated**
    - **Traditional: mean, median...**
      - **“By no means”**
    - **Econometric: inequality indices**

- **Metrics-dependent**
  - **Produce more precise results**
  - **BUT: need to be redone for any new metrics**
  - **Based on fitting probability distributions**

Metrics independent: Coefficient of variation

- **Coefficient of variation: $C = \sigma/\mu$**
  - **Allows to compare distributions with different means**
  - **Sometimes used to assess stability of the metrics**
    - **Metrics is stable for $C < 0.3$**
    - **Unreliable for small samples**
    - **Evolution should be studied…**

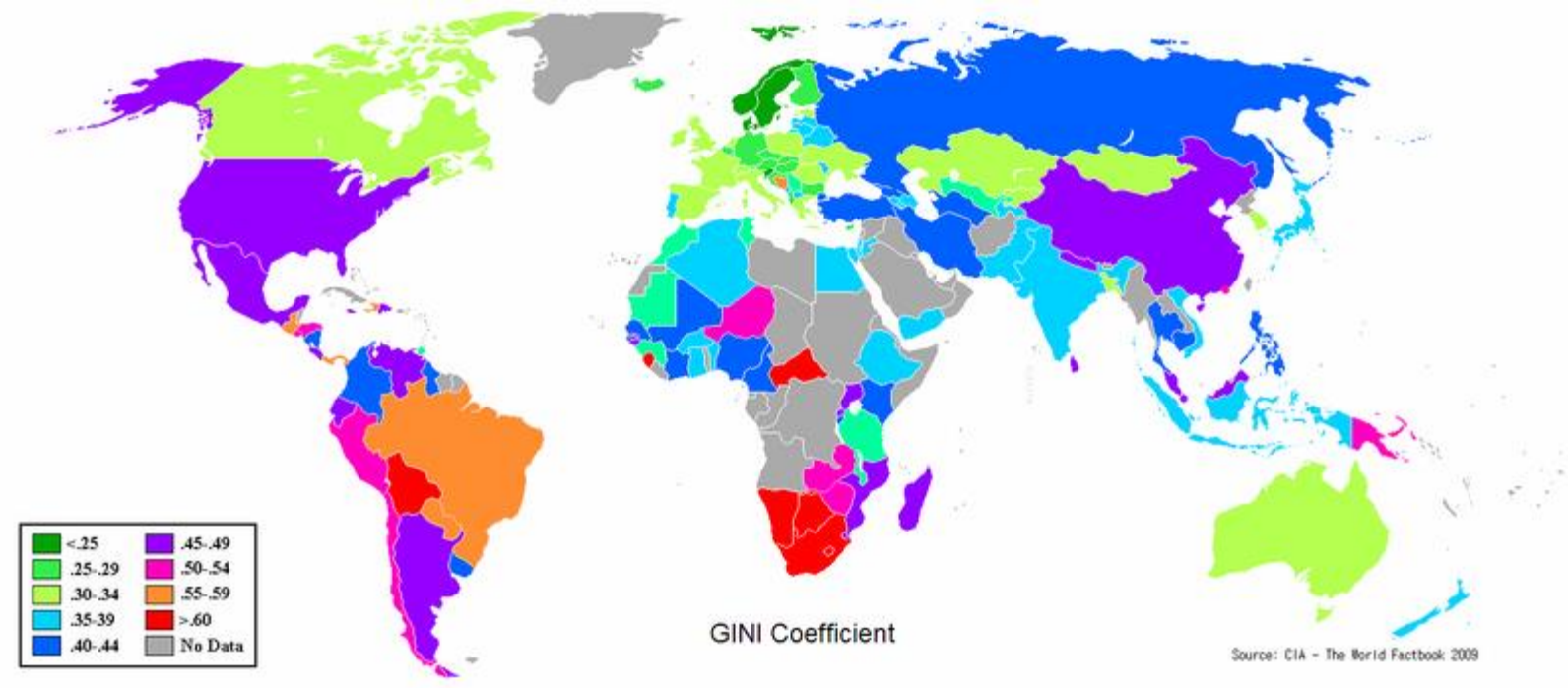
Metrics are like money



- prog. lang.
- domain
- …

- region
- education
- gender
- …

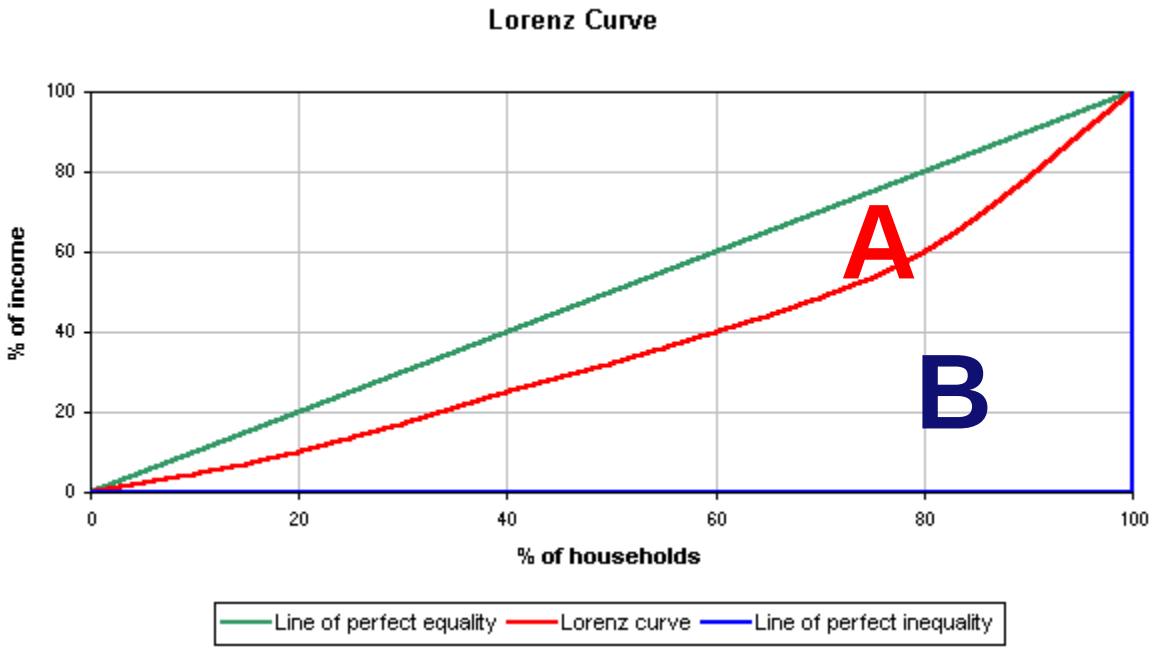
Popular technique: Gini coefficient



- **Gini coefficient measure of economic inequality**
- **Ranges on [0; 1 – 1/n]**
- **High values indicate high inequality**

Gini coefficient: Formally

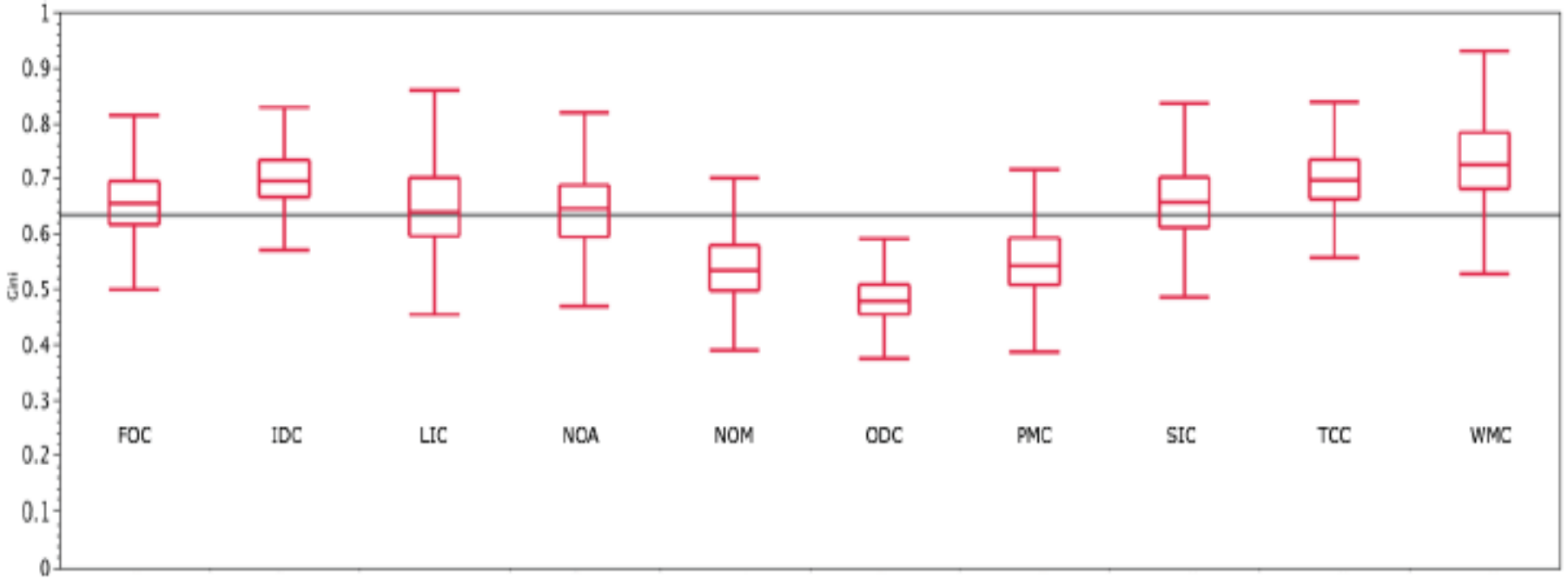
- **Lorenz curve:**
  - **% of income shared by the lower % of the population**



- **Gini = A/(A+B)**
- **Since A+B = 0.5**

  Gini = 2A

Gini and software metrics [Vasa et al. 2009]

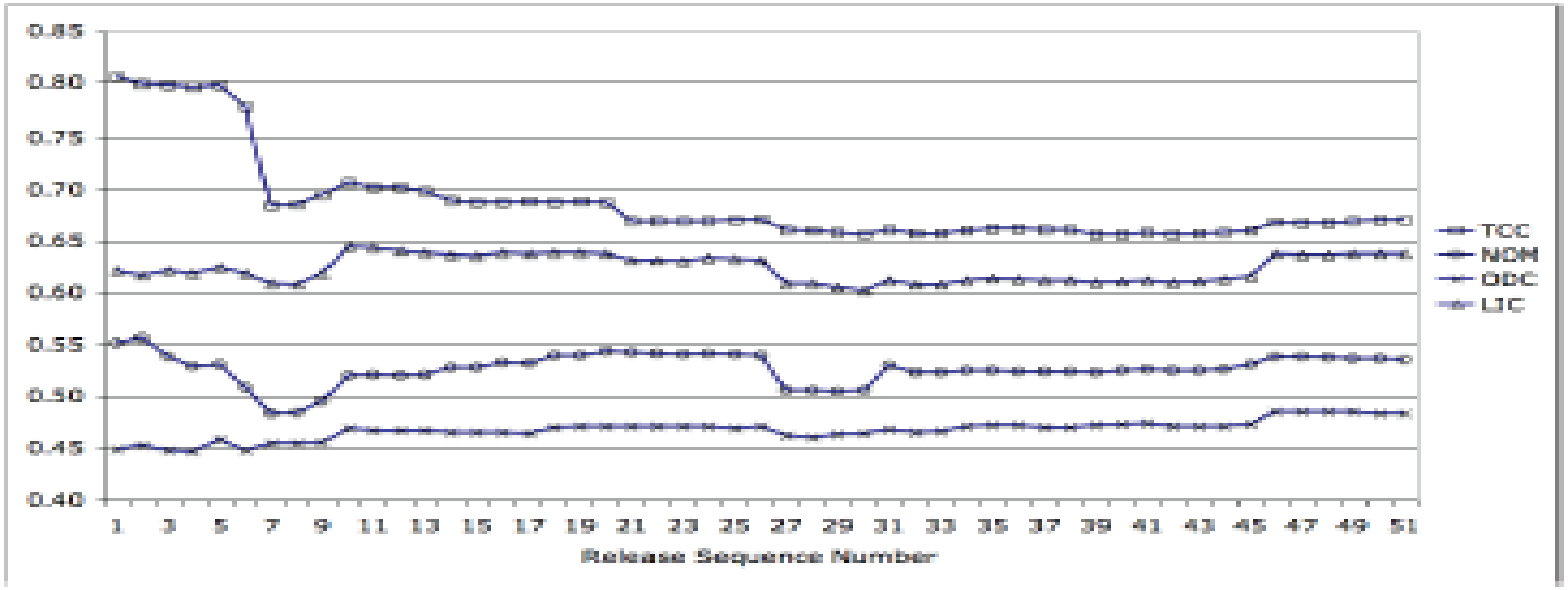


- **For most of the metrics on the benchmark systems: 0.45 ≤ Gini ≤ 0.75**
- **Higher Gini/WMC: presence of generated code or code, structured in a way similar to the generated code (parsers)**

Gini and metrics: Exceptions

System	Metrics	Increase		Explanation
JabRef	WMC	0.75	0.91	Machine generated parser introduced
Checkstyle	Fan-in (classes)	0.44	0.80	Plug-in based architecture introduced.
Jasper-Reports	#Public methods	0.58	0.69	Introduction of a set of new base classes.
WebWork	Fan-out	0.51	0.62	A large utility class and multiple cases of copy-and paste introduced.

Gini and evolution: Spring



- **Rather stable: programmers accumulate competence and tend to solve similar problems by similar means**
- **Similar for other econometric techniques: Theil, Hoover, Atkinson, ...**

Aggregation techniques

- **Metrics-independent**
  - **Applicable for any metrics to be aggregated**
  - **Are the results also metrics-independent?**
  - **Based on econometrics**

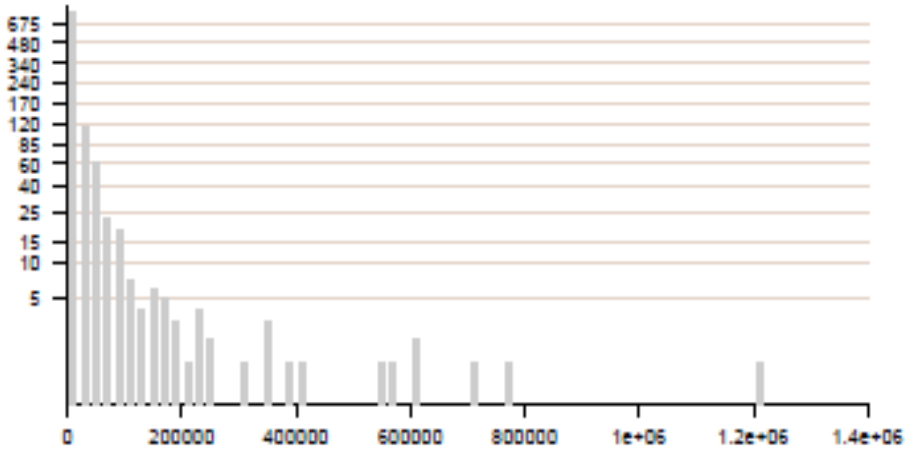
- **Metrics-dependent**
  - **Produces more precise results**
  - **BUT: needs to be redone for any new metrics**
  - **Based on fitting probability distributions**

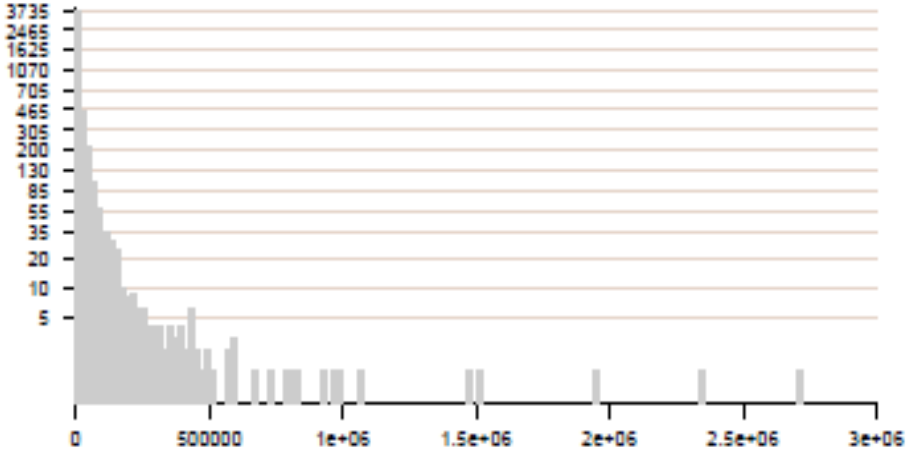
Metrics-dependent aggregation: Statistical fitting

1. Collect the metrics values for the lower-level elements
2. Present a histogram
3. Fit a (theoretical) probability distribution to describe the sample distribution
   a) Select a family of theoretical distributions
   b) Fit the parameters of the probability distribution
   c) Assess the goodness of fit
4. If a theoretical distribution can be fitted, use the fitted parameters as the aggregated value

Step 1: Histograms

- **We have seen quite a number of them already!**

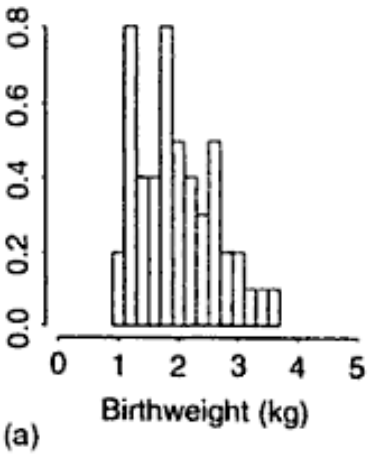


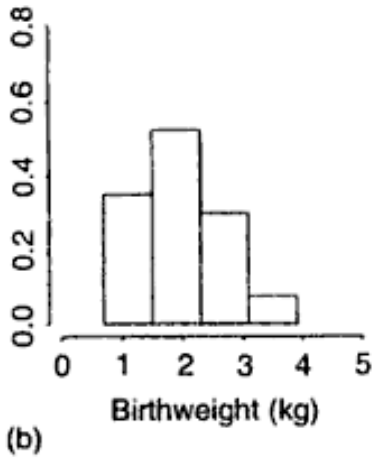


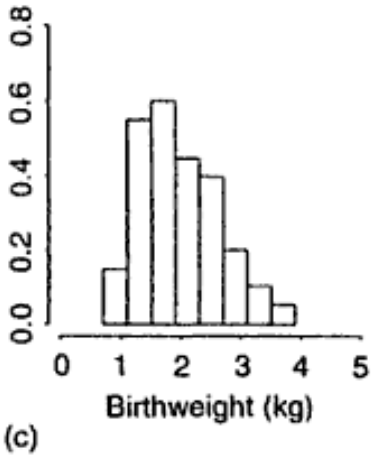
Robles et al. 2006: LOC in Debian 2.0 (left) and 3.0 (right)

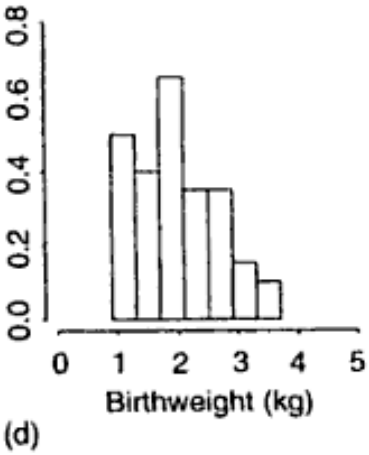
Histograms are not without problems

- **Data: 50 birth weights of children with a severe idiopathic respiratory syndrome**









- **The same data leads to four different "distributions"**
- **What can affect the way histogram looks like?**
  - **Bin width**
  - **Position of the bin's edges**

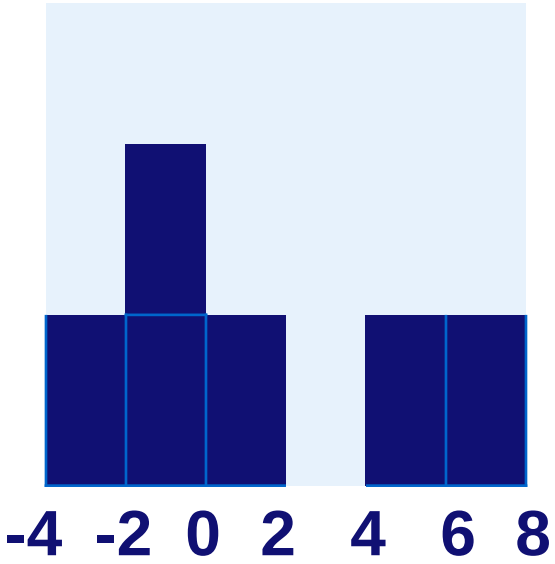
Kernel density estimators

- **Advantages**
  - **Statistically more sound (no dependency on the end-points of the bins)**
  - **Produces smooth curves**

- **Disadvantages**
  - **Statistically more complex**
  - **Parameter tuning might be a challenge**

Kernel density estimates: Intuition

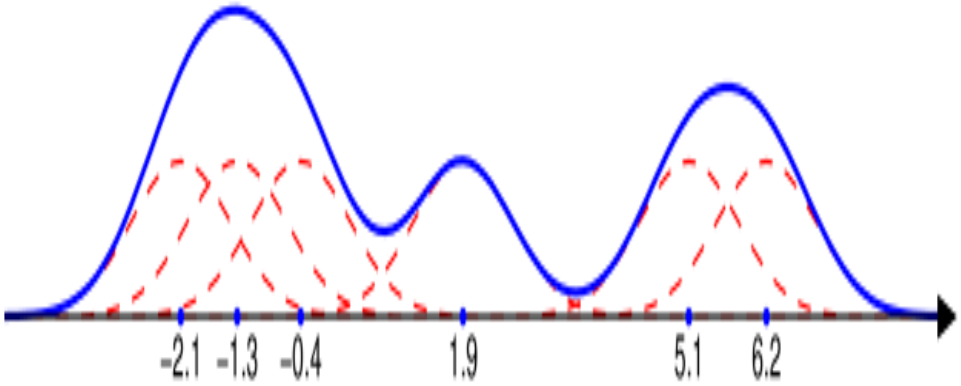
- **Data: -2.1, -1.3, -0.4, 1.9, 5.1, 6.2**



Histogram: every value is a rectangle.

Shape is a "sum" of the rectangles.

What if each value will be a "bump" that can be added together to create a smooth curve?

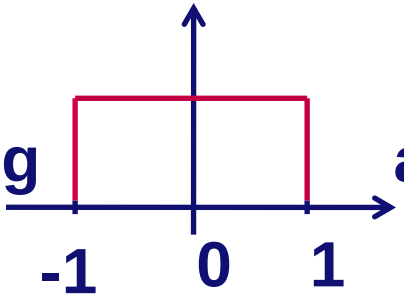


Kernel density estimation: Formally

$$f(x) = \frac{1}{nh} \sum_{i=1}^{n} K\left(\frac{x - x_i}{h} \right)$$

Where

- **n – number of observations**
- **h – a smoothing parameter, the "bandwidth"**
- **K – a weighting function, the "kernel"**

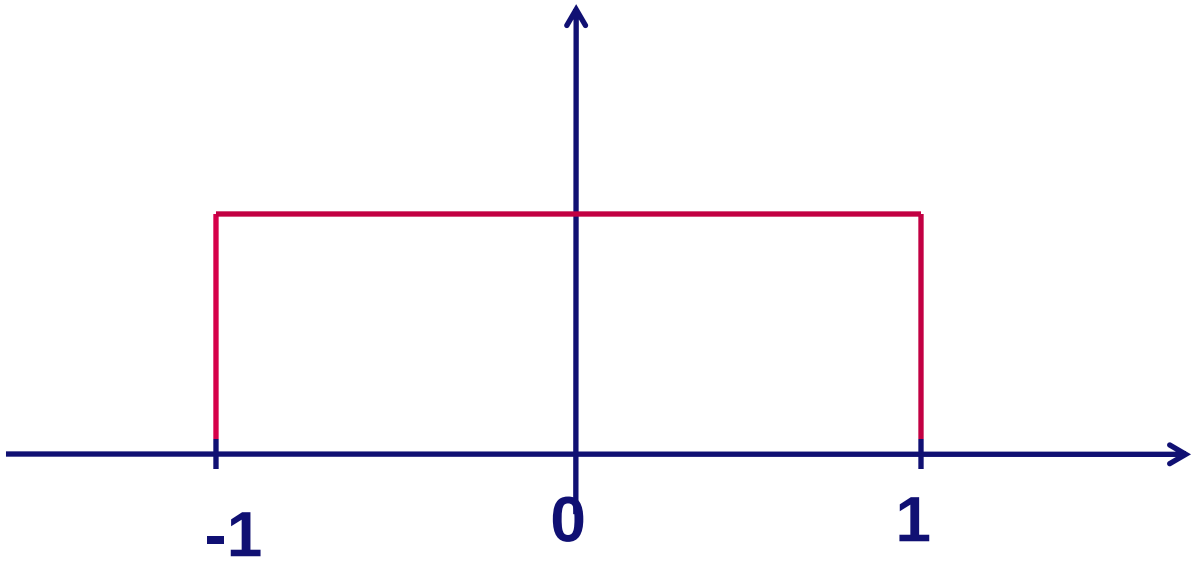
Histogram can be obtained using 

as K

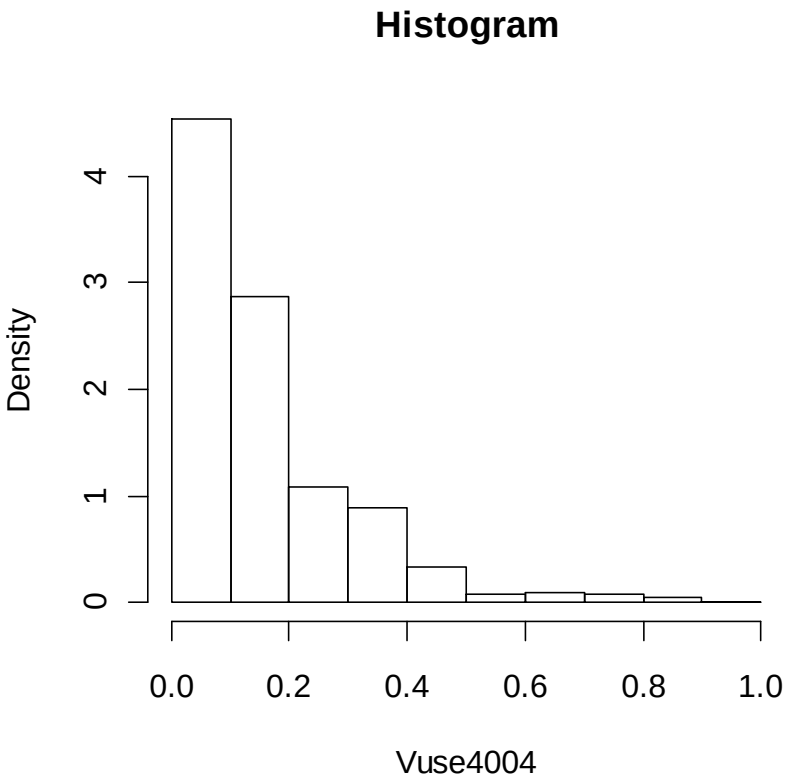
Once K is chosen one can determine the optimal h.

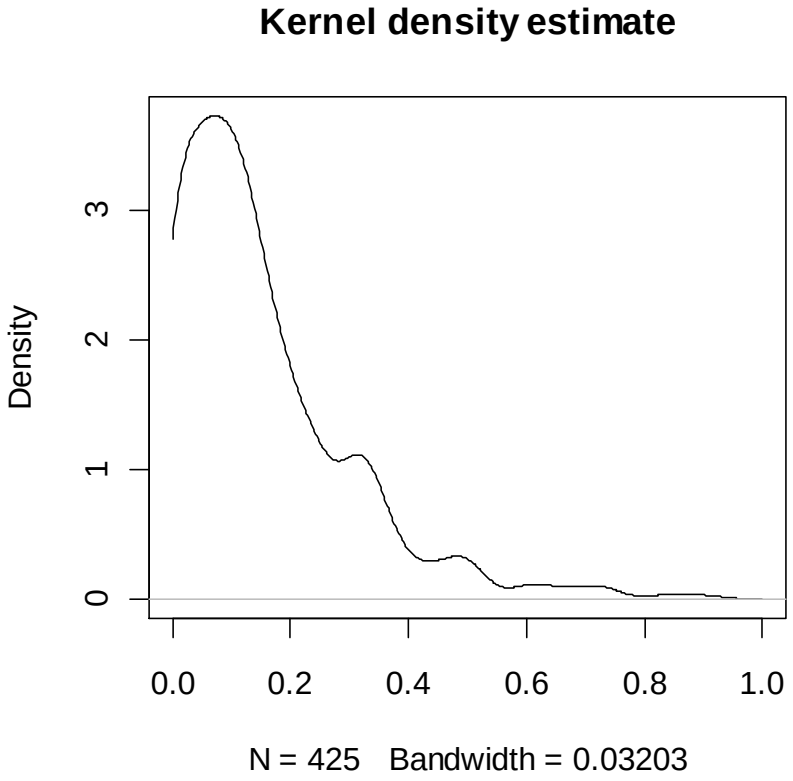
Histogram as a kern density estimate

- **h is the bin width**
- **x and x_i are in the same bin if** $-1 \leq \frac{x - x_i}{h} \leq 1$

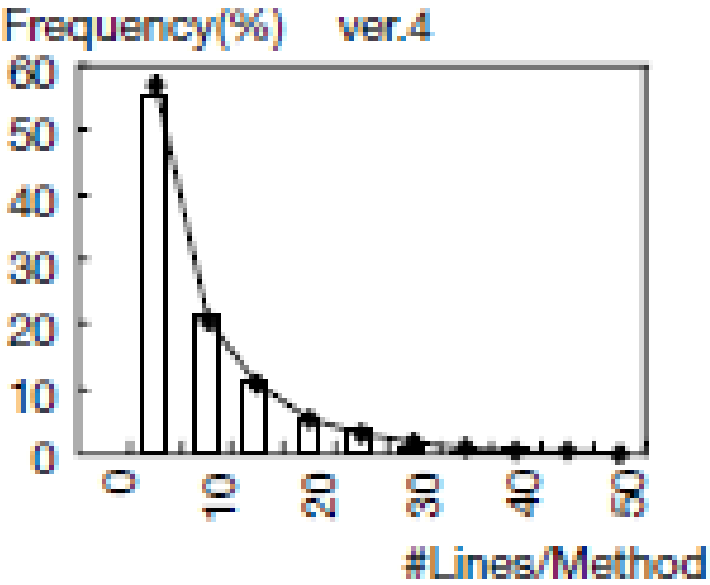


Histogram vs. Kernel density estimate

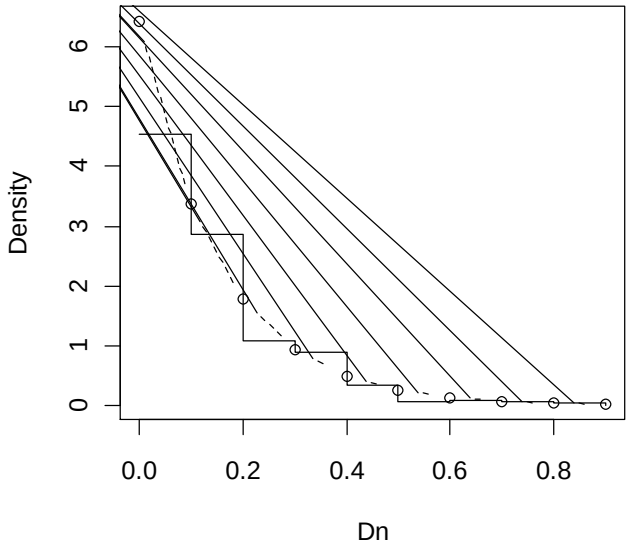




Step 2: fitting a distribution



Tamai, Nakatani.
Negative binomial distribution



S, Roubtsov, vd Brand
Exponential distribution

- **Family of distributions is chosen based on shape**
- **If the parameters fitting is not good enough try a different one!**

Step 3c. Goodness of fit: Pearson χ^2 test

- **The test statistic**

$$X^2 = \sum_{i=1}^{n} \frac{(O_i - E_i)^2}{E_i}$$

where

- **O – observed frequency of the result i**
- **E – expected frequency of the result i**

- **Compare X^2 with the theoretical χ^2 distribution for the given number of degrees of freedom: $P(\chi^2 > X^2)$**
  - **Degrees of freedom = number of observations – number of fitted parameters**
  - **Comparison is done based on table values**
  - **If the $P(\chi^2 > X^2) <$ threshold – the fit is good**
  - **Common thresholds are 0.1, 0.05 and 0.01**

Recapitulation: Statistical fitting

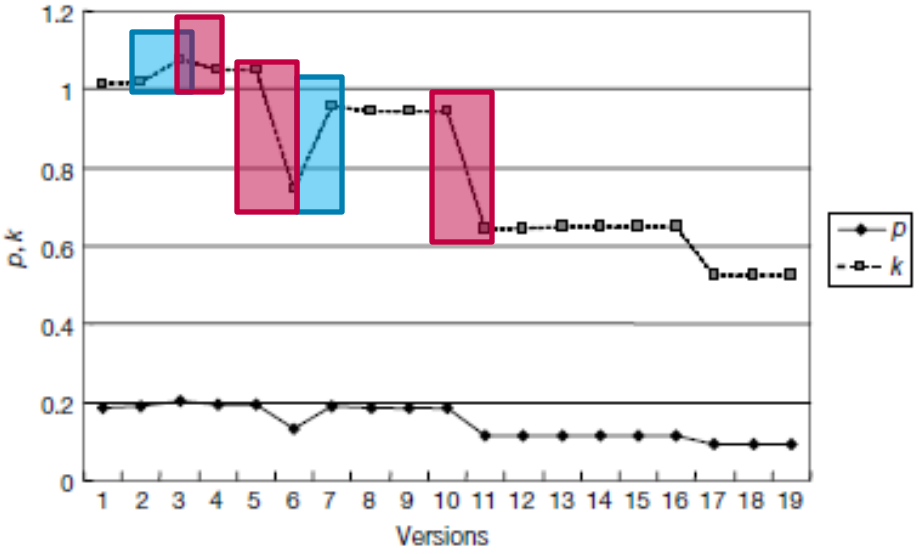
1. Collect the metrics values for the lower-level elements
2. Present a histogram
3. Fit a (theoretical) probability distribution to describe the sample distribution
   a) Select a family of theoretical distributions
   b) Fit the parameters of the probability distribution
   c) Assess the goodness of fit
4. If a theoretical distribution can be fitted, use the fitted parameters as the aggregated value

What about the evolution of the aggregated values?

- Geometry library: Jun, subsystem “Geometry”
- Tamai, Nakatani: Negative binomial distribution

$$f(x) = \binom{x-1}{k-1} p^k (1-p)^{x-k}$$

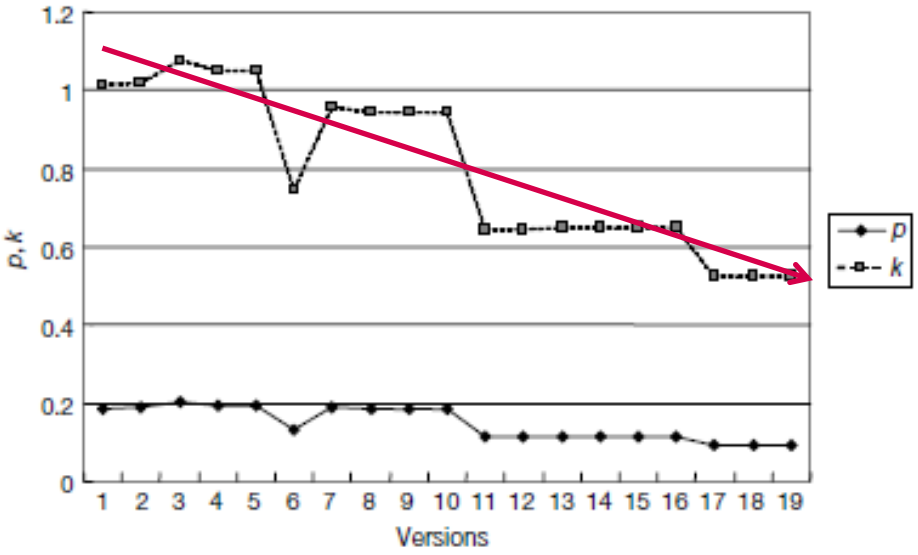
- p, k – distribution parameters
- $\binom{x-1}{k-1}$ - binomial coefficient extended to the reals



- Increase – functionality enhancement
- Decrease – refactoring

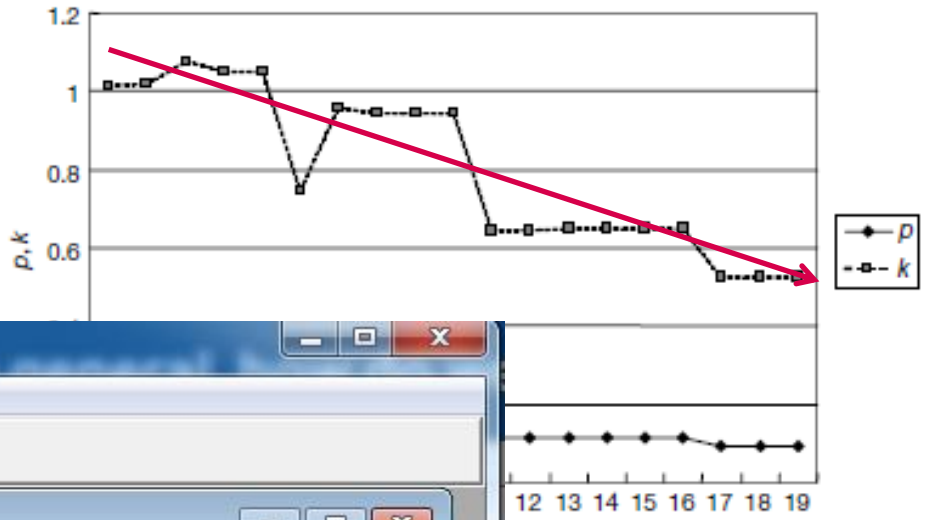
In general, how do we study evolution?

- **Visual inspection**
  - **Is this a real "trend" or just noise?**



In general, how do we study evolution?

- **Time-series analysis**
  - **Simplest form: linear regression with *time***



RGui

File Edit View Misc Packages Windows Help

R Console

```
> cor.test(data,time)

        Pearson's product-moment correlation

data:  data and time
t = -7.4684, df = 16, p-value = 1.337e-06
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 -0.9552531 -0.7046643
sample estimates:
       cor
-0.8815246
```

Linear trend

Significant

Strong

More advanced techniques:

2DD23 - Time series analysis and forecasting

Summary

- **Aggregation:**
  - **Metrics-independent**
    - **Applicable for any metrics to be aggregated**
      - **Traditional: mean, median...**
        - **“By no means”**
      - **Econometric: inequality indices**
  - **Metrics-dependent**
    - **Produce more precise results**
    - **BUT: need to be redone for any new metrics**
    - **Based on fitting probability distributions**

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

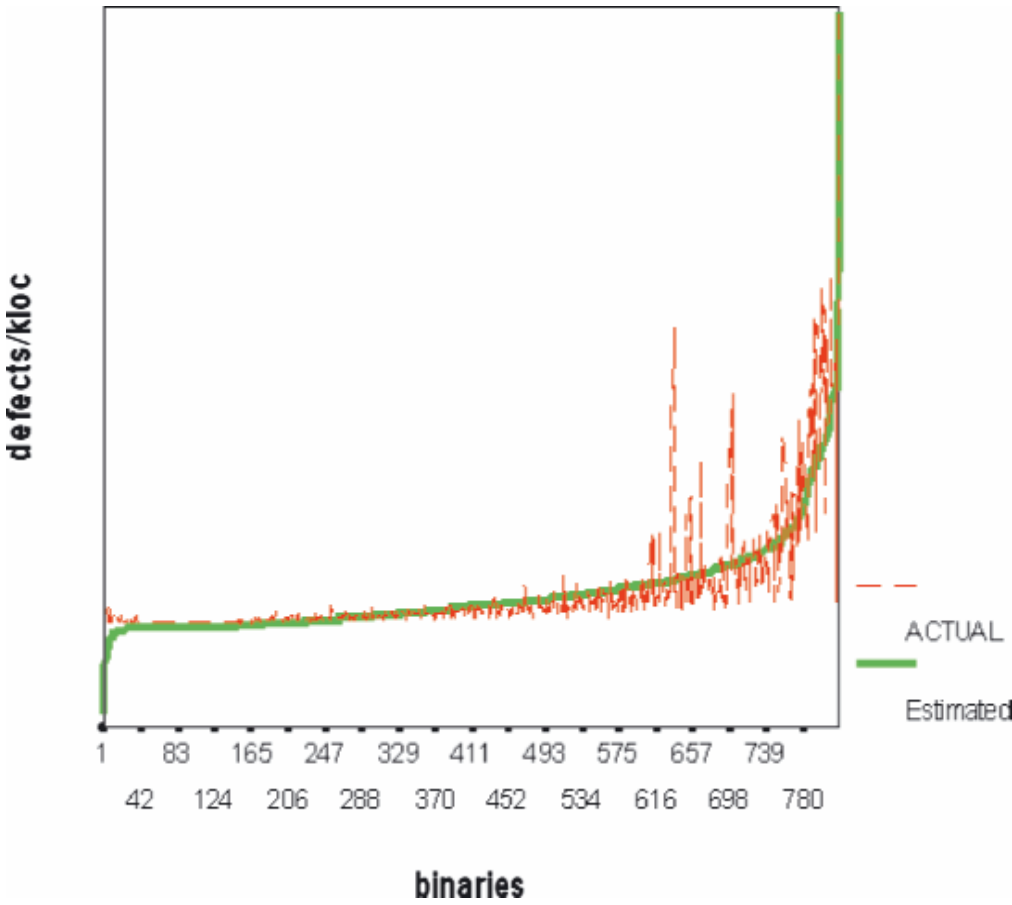
- **Analyze Code Churn between WS2003 and WS2003-SP1 to predict defect density in WS2003-SP1**
  - **40 million LOC, 2000 binaries**
  - **Use absolute and relative churn measures**

- **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**