2IMP25 Software Evolution

Software metrics

Alexander Serebrenik

Technische Universiteit
Eindhoven
University of Technology

Where innovation starts

Assignment 3

- **Ineallation instructions are on Peach**
  - **Ubuntu: thank you, Don!**
  - **Windows: thank you, Adrian and Nathan!**

Metrics of software process

- How much will it cost us to build the system?
- How much effort has been spent on building the system?

- Effort estimation techniques
  - Size-based
  - Complexity-based
  - Functionality-based
  - More advanced techniques are known but go beyond the topics of this class

Size-based effort estimation

- **Estimation models:**
  - **In: SLOC (estimated)**
  - **Out: Effort, development time, cost**

  - **Usually use "correction coefficients" dependent on**
    - **Manually determined categories of application domain, problem complexity, technology used, staff training, presence of hardware constraints, use of software tools, reliability requirements…**
    - **Correction coefficients come from tables based on these categories**
    - **Coefficients were determined by multiple regression**
- **Popular (industrial) estimation model: COCOMO**

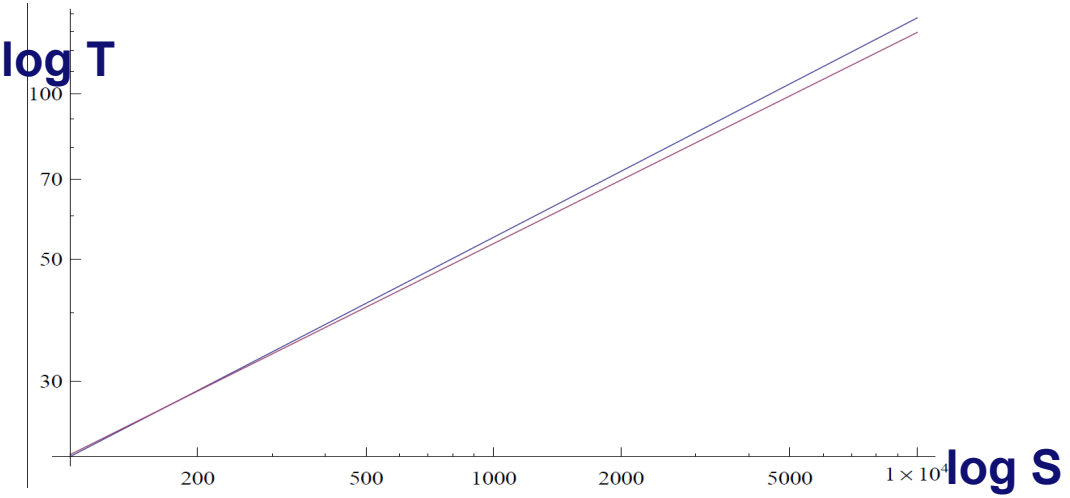
Basic COCOMO

$$E = aS^b$$

$$T = cE^d$$

	a	b	c	d
Information system	2.4	1.05	2.5	0.38
Embedded system	3.6	1.20	2.5	0.32

- **E – effort (man-months)**
- **S – size in KLOC**
- **T – time (months)**
- **a, b, c and d – correctness coefficients**



More advanced COCOMO: even more categories

Advanced COCOMO

Software Size Sizing Method: Source Lines of Code

	SLOC	% Design Modified	% Code Modified	% Integration Required	Assessment and Assimilation (0% - 8%)	Software Understanding (0% - 50%)	Unfamiliarity (0-1)
New							
Reused		0	0				
Modified							

Software Scale Drivers

Precedentedness	Nominal	Architecture / Risk Resolution	Nominal	Process Maturity	Nominal
Development Flexibility	Nominal	Team Cohesion	Nominal		

Software Cost Drivers

Product

Required Software Reliability	Nominal
Data Base Size	Nominal
Product Complexity	Nominal
Developed for Reusability	Nominal
Documentation Match to Lifecycle Needs	Nominal

Personnel

Analyst Capability	Nominal
Programmer Capability	Nominal
Personnel Continuity	Nominal
Application Experience	Nominal
Platform Experience	Nominal
Language and Toolset Experience	Nominal

Platform

Time Constraint	Nominal
Storage Constraint	Nominal
Platform Volatility	Nominal

Project

Use of Software Tools	Nominal
Multisite Development	Nominal
Required Development Schedule	Nominal

Software Labor Rates

Cost per Person-Month (Dollars)

Calculate

siteit

ology

Complexity-based effort estimation

- **Do you recall Halstead?**
- **Effort: E = V * D**
  - **V – volume, D – difficulty**

$$E = (N_1 + N_2)\ln(n_1 + n_2) * \frac{n_1}{2} * \frac{N_2}{n_2}$$

  - **Potentially problematic: questioned by Fenton and Pfleger in 1997**
- **Time to understand/implement (sec): T = E/18**

Code is not everything

- **Lehman 6:**
  - **The functional capability <…> must be continually enhanced to maintain user satisfaction over system lifetime.**

- **How can we measure amount of functionality in the system?**
  - **[Albrecht 1979] "Function points"**
    - **Anno 2015: Different variants: IFPUG, NESMA, …**
  - **Determined based on system description**
  - **Amount of functionality can be used to assess the development effort and time before the system is built**
  - **Originally designed for information systems**

Functionality and effort

What kinds of problems could have influenced validity of this data?

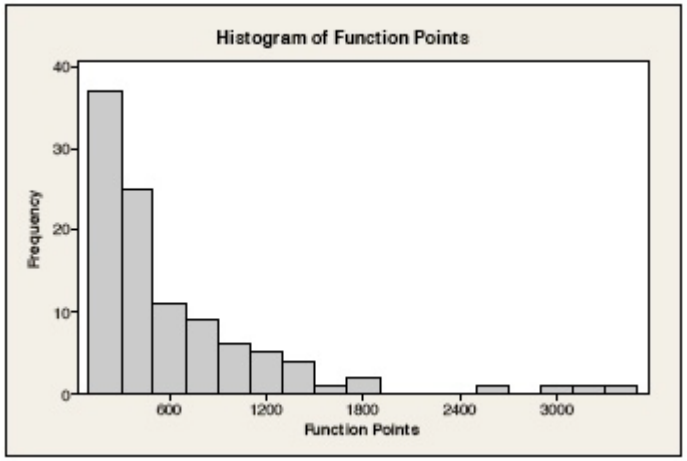
TABLE 3-34 U.S. Average Productivity for Selected Methodologies (Data Expressed in Function Points per Staff Month)

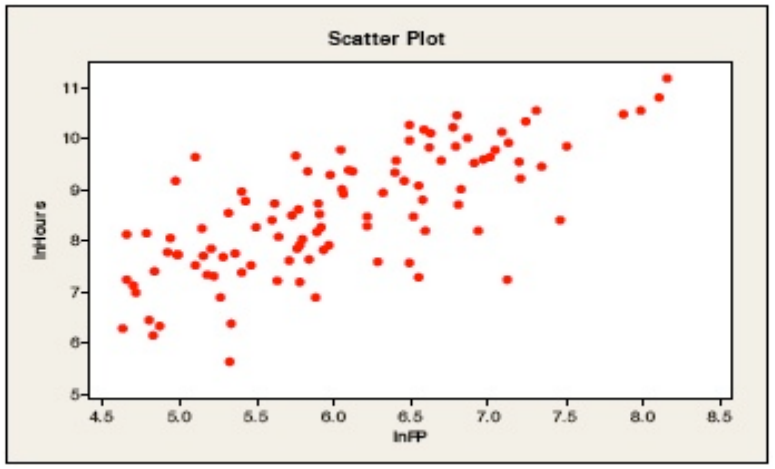
Methodology	100	1,000	10,000	100,000	Average
Agile	37.00	25.00	7.50	— No data	23.17
CMM 1	12.00	7.20	4.50	1.70	6.35
CMM 3	21.00	10.20	6.30	4.80	10.58
CMM 5	23.00	12.60	8.90	6.50	12.75
Extreme (XP)	35.00	23.60	8.60	— No data	22.40
Iterative	19.00	9.90	6.70	4.20	9.95
Object-oriented	20.00	16.70	7.70	5.30	12.43
Six-Sigma	17.50	11.00	8.20	6.30	10.75
TSP/PSP	16.00	18.00	9.20	6.85	12.51
Waterfall	8.80	6.50	4.2	1.80	5.33
Average	**20.93**	**14.07**	**7.14**	**4.16**	**11.58**

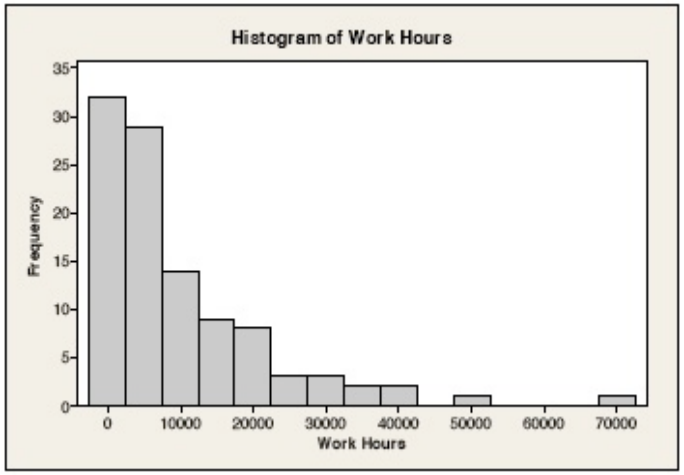
< 10% US comp.

Functionality and effort

- **104 projects at AT&T from 1986 through 1991**





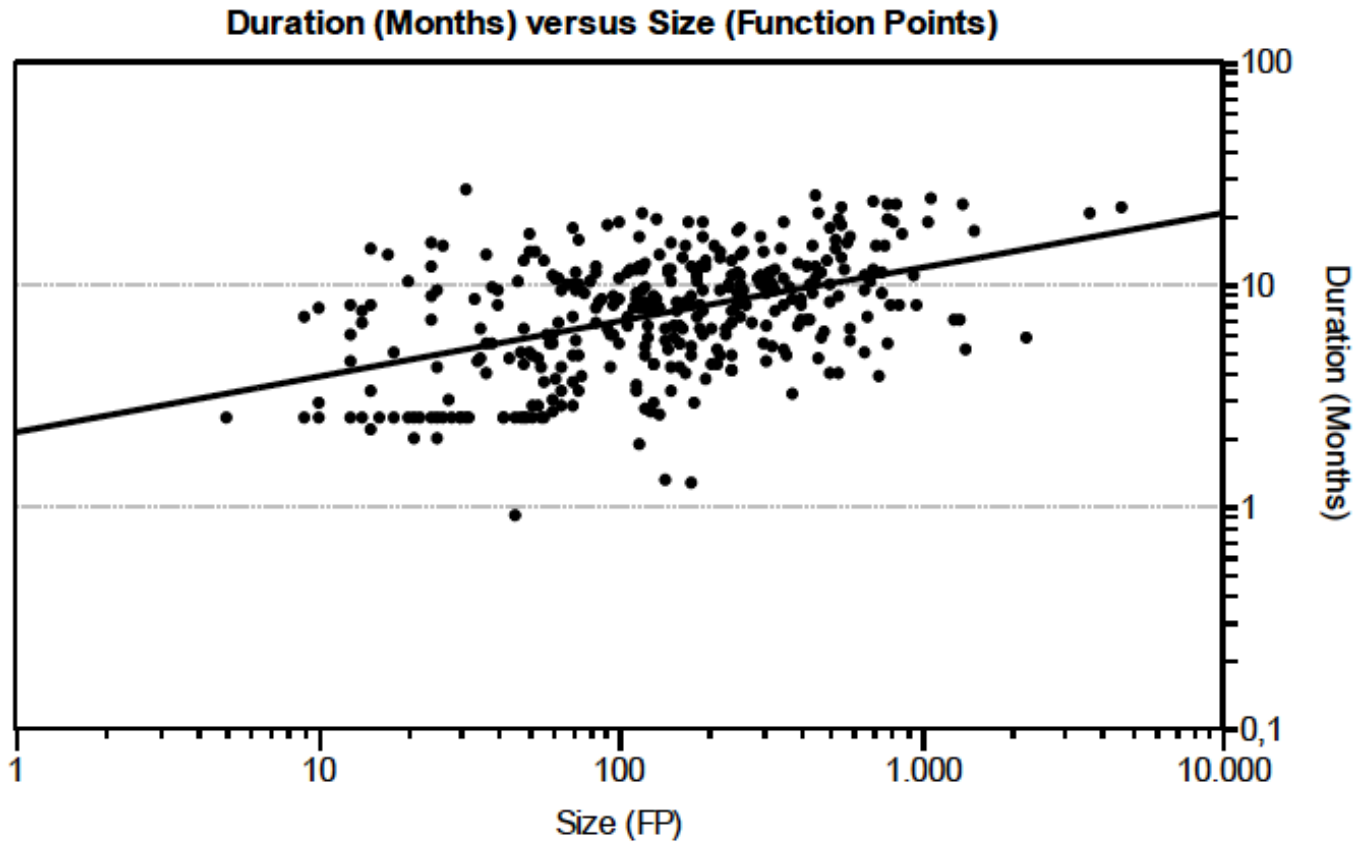


$$\ln(E_{est}) = 2.5144 + 1.0024\ln(FP)$$

What about the costs?

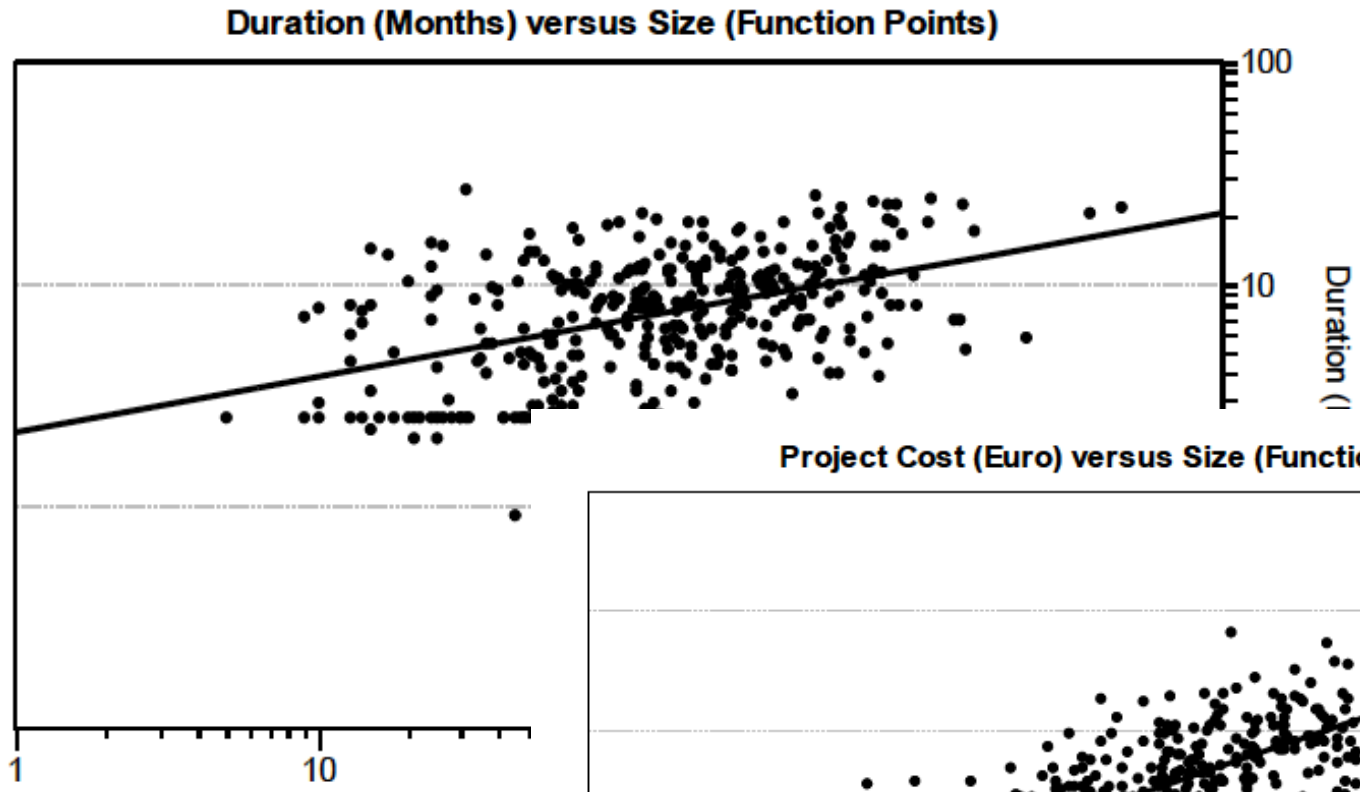
Function points	Cost per fp
1	---
10	---
100	$795.36
1000	$1136.36
10000	$2144.12
100000	$3639.74

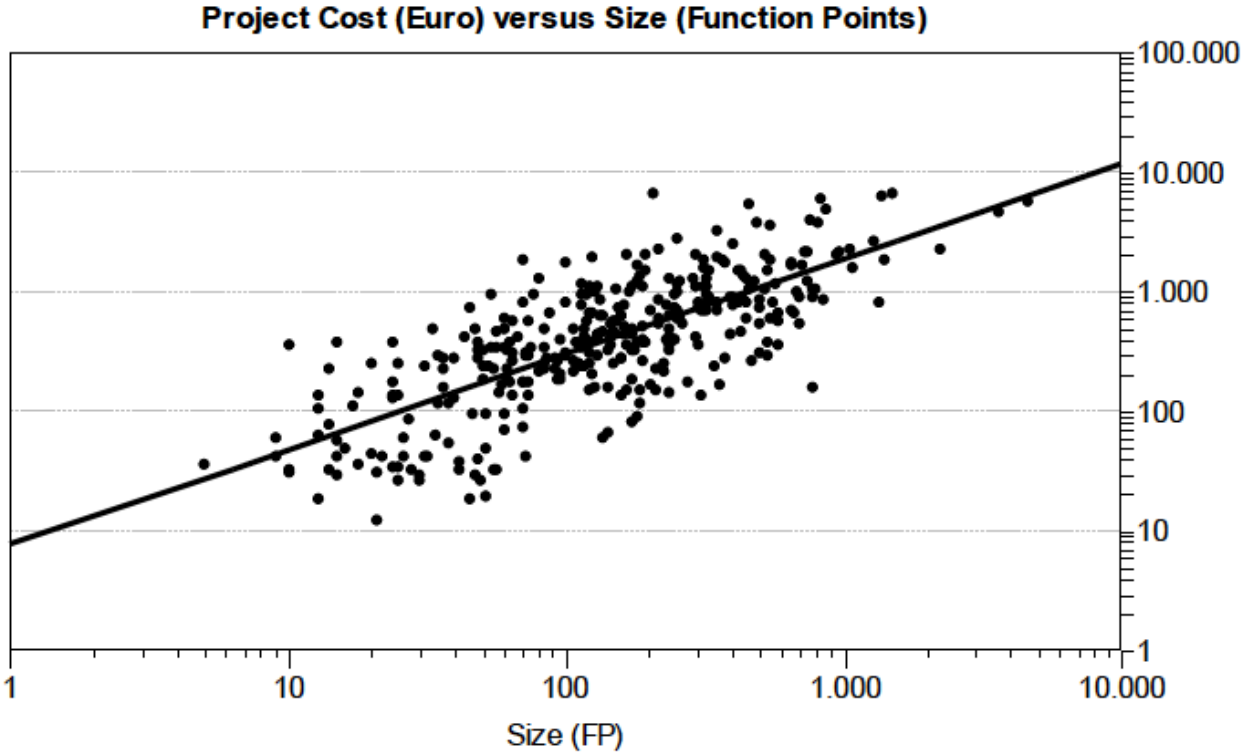
Function points, duration, cost…



[Huijgens, van Solingen, van Deursen 2014]

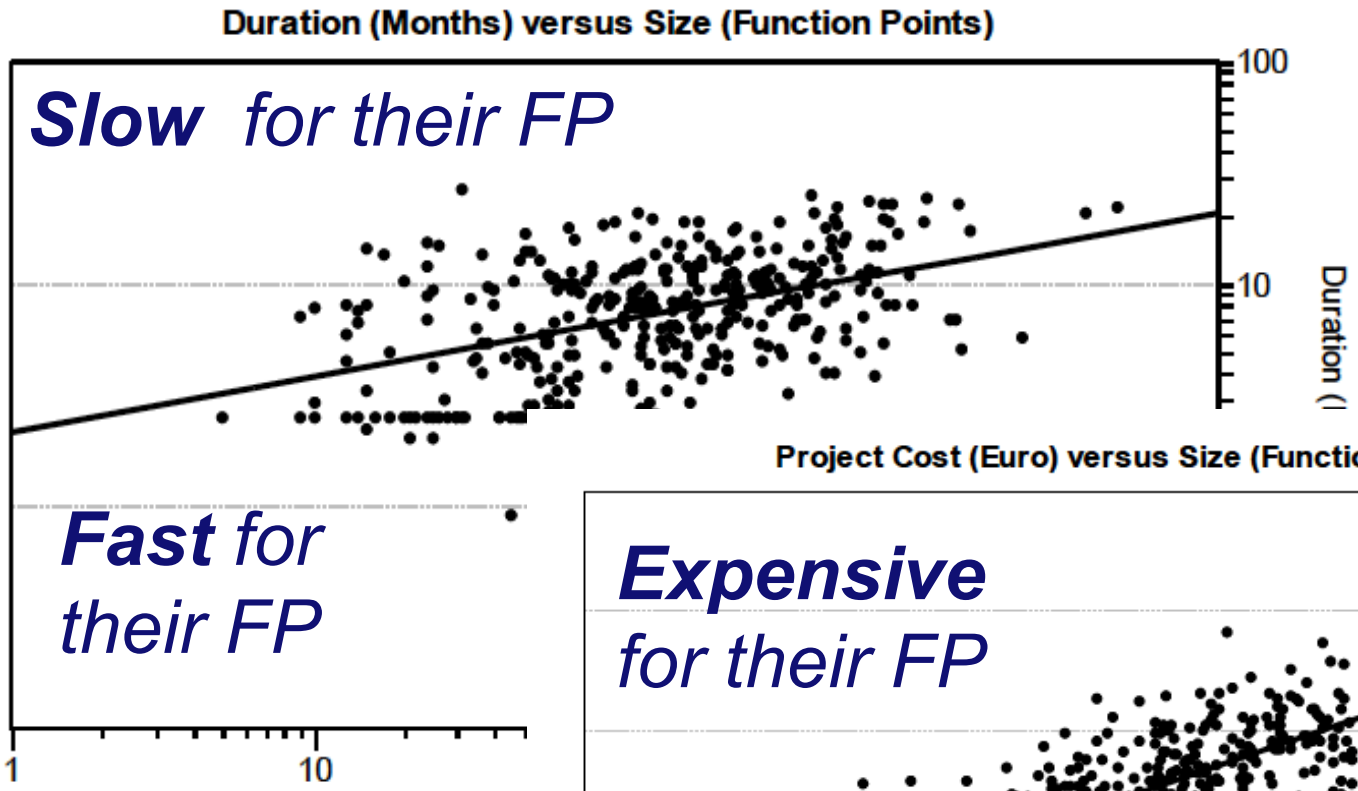
Function points, duration, cost…

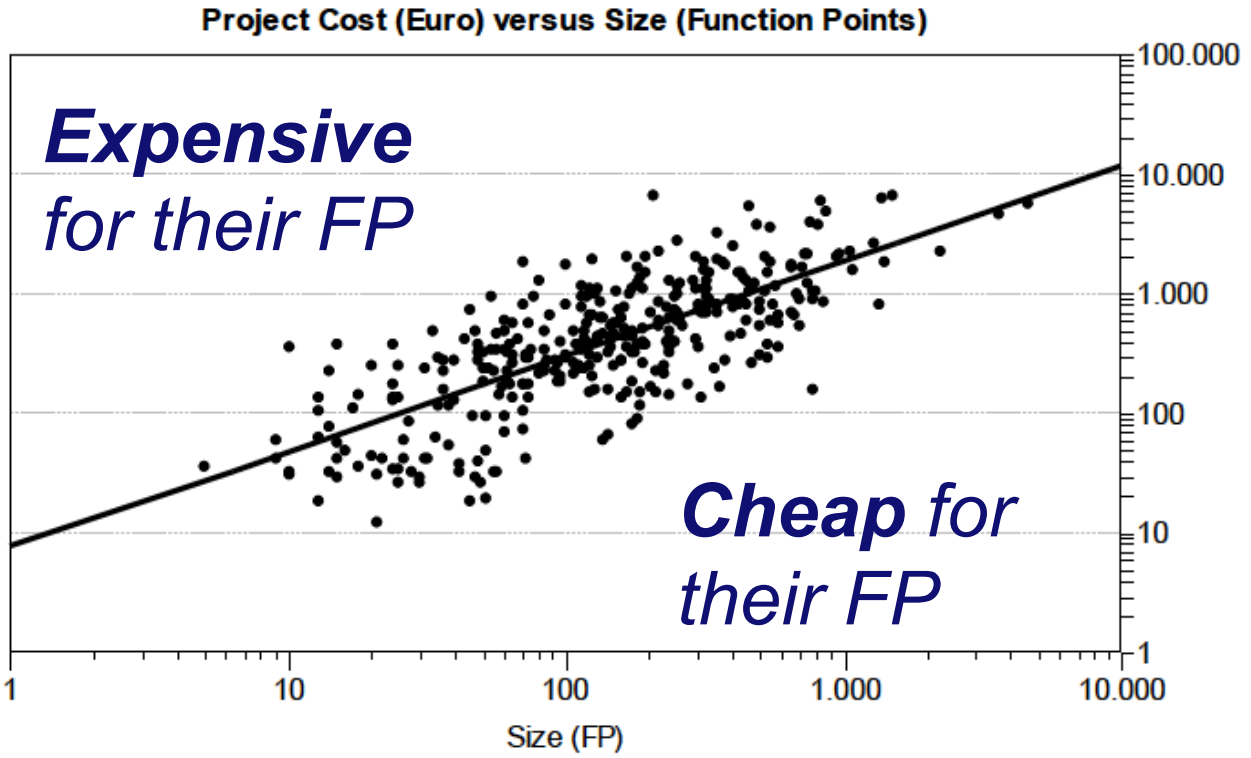




[Huijgens, van Solingen, van Deursen 2014]

Function points, duration, cost…



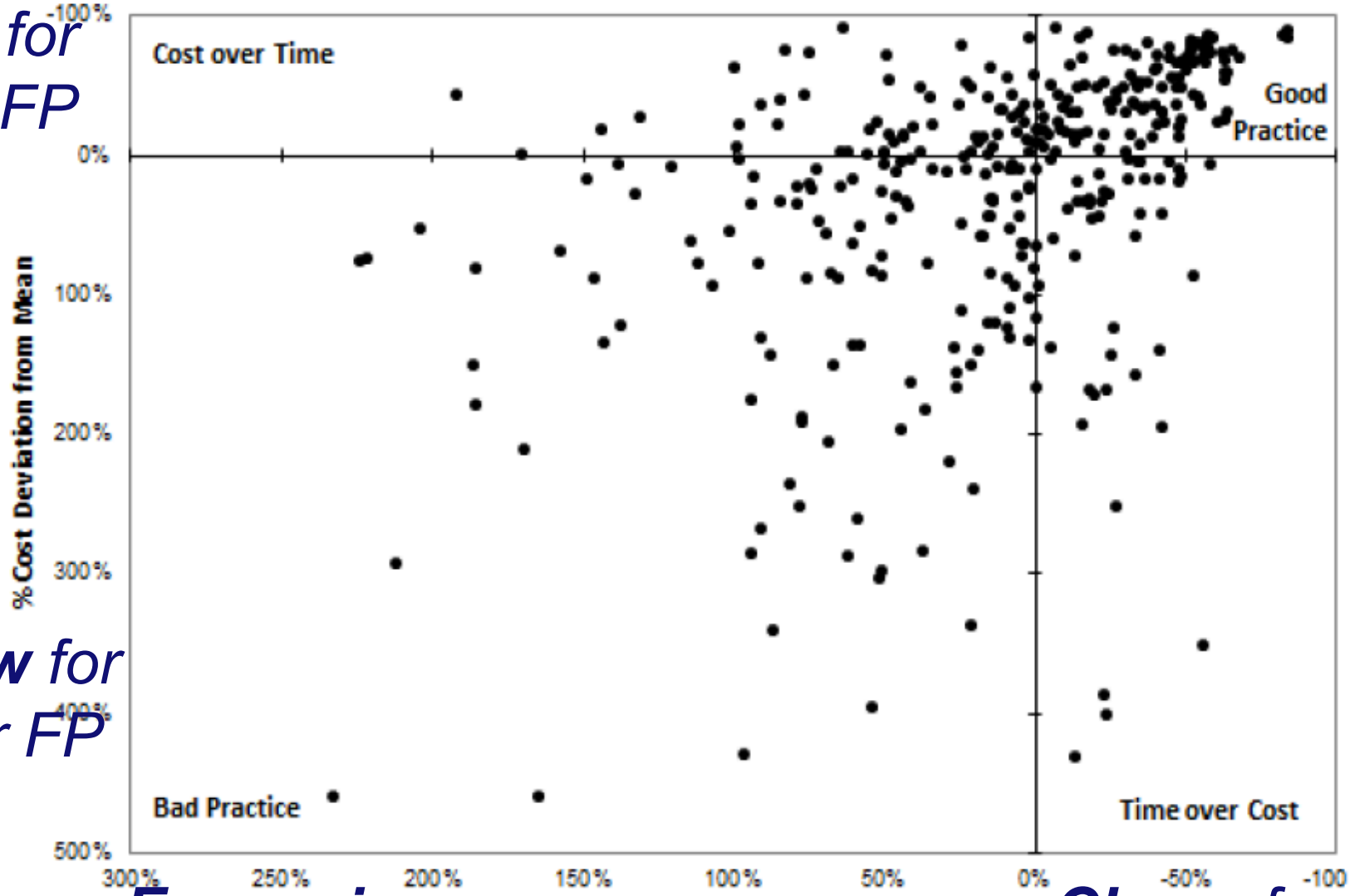


[Huijgens, van Solingen, van Deursen 2014]

Cost/Duration matrix [Huijgens, v Solingen, v Deursen 2014]

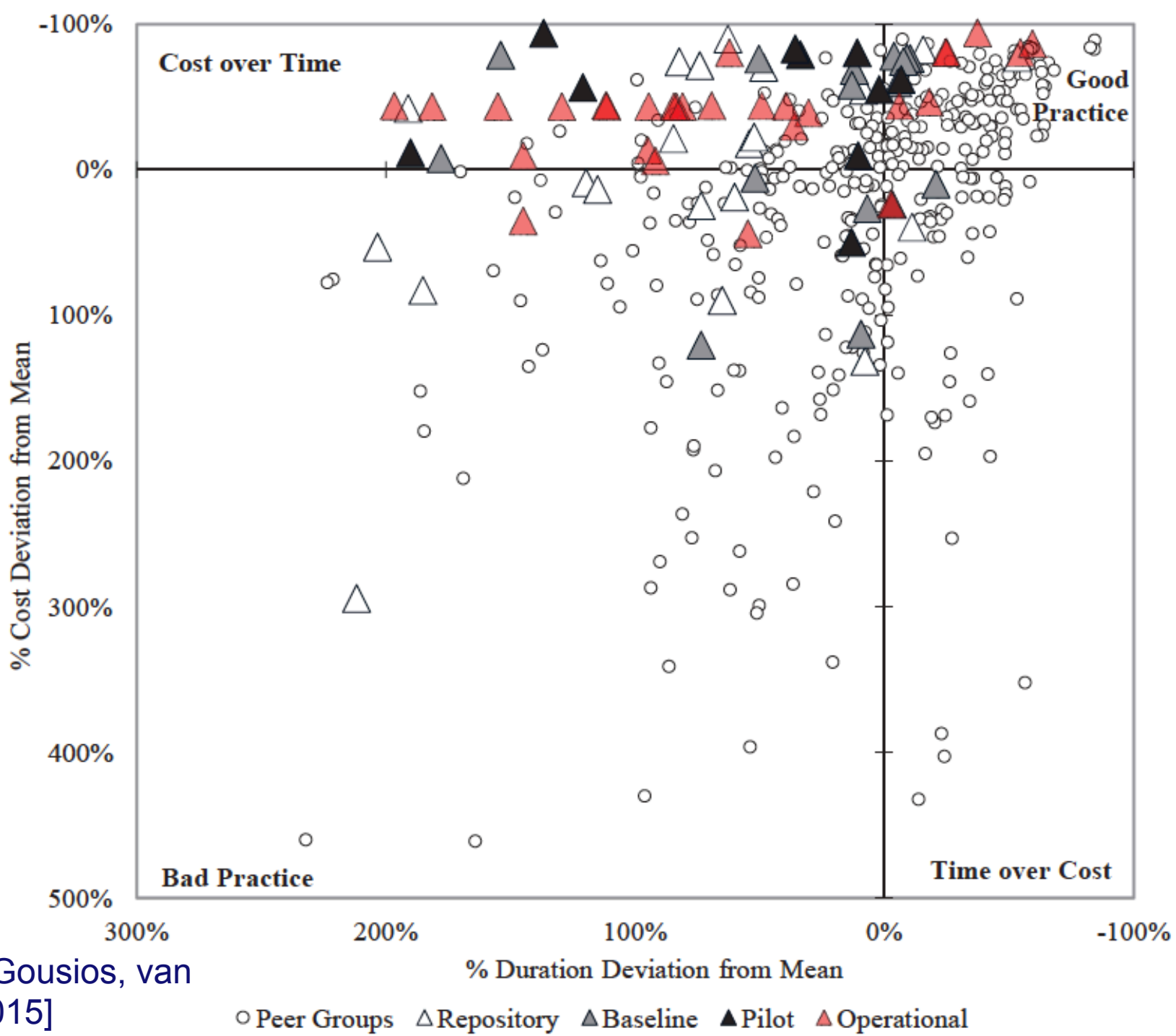
***Fast** for their FP*

***Slow** for their FP*

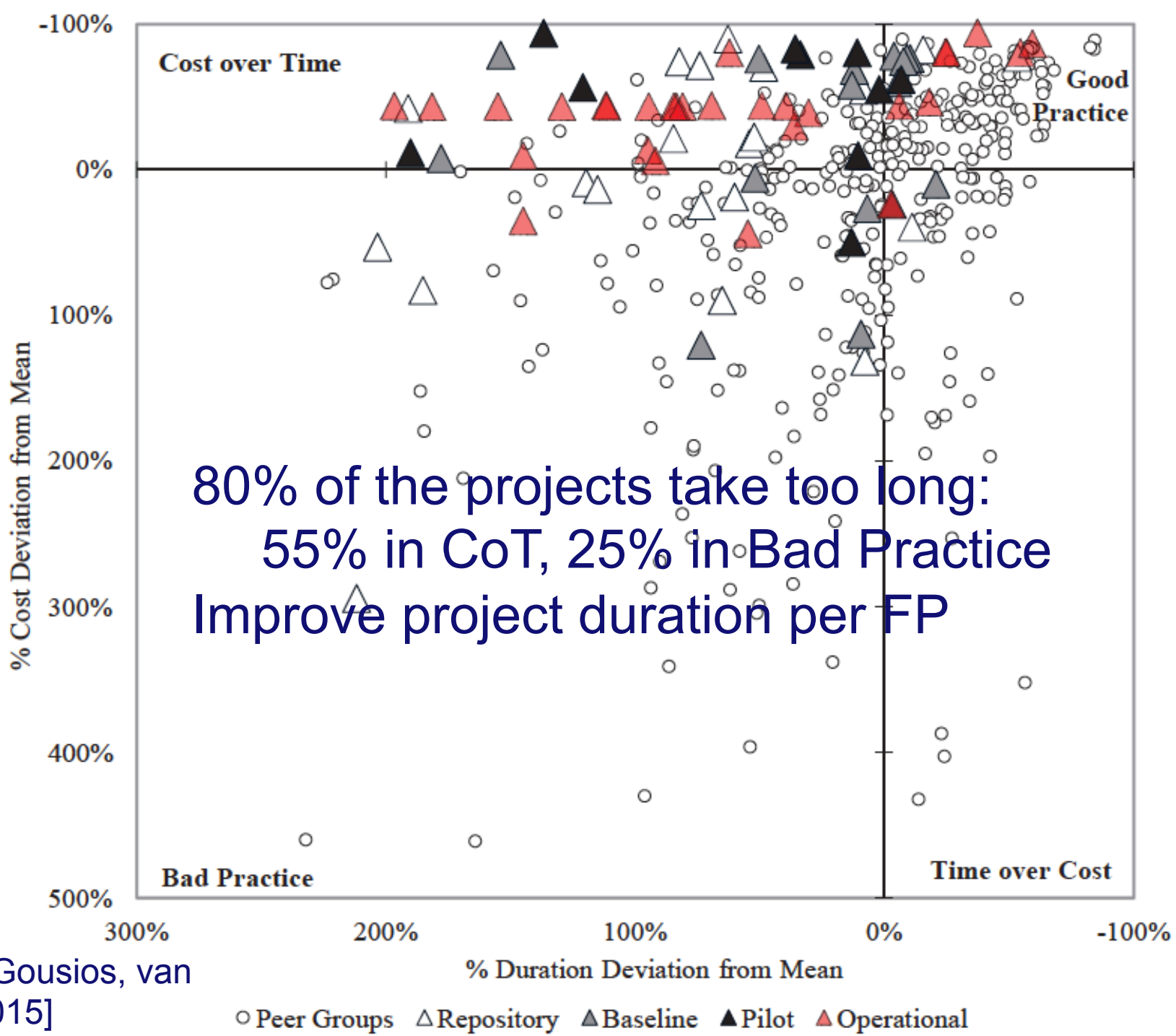


***Expensive** for their FP*

***Cheap** for their FP*



[Huijgens, Gousios, van Deursen 2015]



80% of the projects take too long:
55% in CoT, 25% in Bad Practice
Improve project duration per FP

[Huijgens, Gousios, van Deursen 2015]

How to determine the number of function points? [IFPUG original version]

- **Identify primitive constructs:**
  - **inputs: web-forms, sensor inputs, mouse-based, …**
  - **outputs: data screens, printed reports and invoices, …**
  - **logical files: table in a relational database**
  - **interfaces: a shared (with a different application) database**
  - **inquiries: user inquiry without updating a file, help messages, and selection messages**

TABLE 2-15 The Initial IFPUG Version of the Function Point Metric

Significant Parameter	Low Complexity	Medium Complexity	High Complexity
External input	×3	×4	×6
External output	×4	×5	×7
Logical internal file	×7	×10	×15
External interface file	×5	×7	×10
External inquiry	×3	×4	×6

Software is not only functionality!

- **Non-functional requirement necessitate extra effort**
  - **Every factor on [0;5]**
  - **Sum * 0.01 + 0.65**
  - **Result * Unadjusted FP**
- **1994: Windows-based spreadsheets or word processors: 1000 – 2000**

C1	Data communications
C2	Distributed functions
C3	Performance objectives
C4	Heavily used configuration
C5	Transaction rate
C6	On-line data entry
C7	End-user efficiency
C8	On-line update
C9	Complex processing
C10	Reusability
C11	Installation ease
C12	Operational ease
C13	Multiple sites
C14	Facilitate change

Function points, effort and development time

- **Function points can be used to determine the development time, effort and ultimately costs**
  - **Productivity tables for different SE activities, development technologies, etc.**

- **Compared to COCOMO**
  - **FP is applicable for systems to be built**
    - **COCOMO is not**
  - **COCOMO is easier to automate**
  - **Popularity:**
    - **FP: information systems, COCOMO: embedded**

But what if the system already exists?

- We need it, e.g., to estimate maintenance or reengineering costs

- Approaches:
  - Derive requirements ("reverse engineering") and calculate FP based on the requirements derived
  - Jones: Backfiring
    - Calculate LLOC (logical LOC, source statements)
    - Divide LLOC by a language-dependent coefficient

    - What is the major theoretical problem with backfiring?

Backfiring in practice

- **What can you say about the precision of backfiring?**
  - **Best: ± 10% of the manual counting**
  - **Worst: +100% !**
- **What can further affect the counting?**
  - **LOC instead of LLOC**
  - **Generated code, …**
  - **Code and functionality reuse**

		Source Statements per Function Point		
Language	Nominal Level	Low	Mean	High
1st Generation	1.00	220	320	500
Basic assembly	1.00	200	320	450
Macro assembly	1.50	130	213	300
C	2.50	60	128	170
BASIC (interpreted)	2.50	70	128	165
2nd Generation	3.00	55	107	165
FORTRAN	3.00	75	107	160
ALGOL	3.00	68	107	165
COBOL	3.00	65	107	150
CMS2	3.00	70	107	135
JOVIAL	3.00	70	107	165
PASCAL	3.50	50	91	125
3rd Generation	4.00	45	80	125
PL/I	4.00	65	80	95
MODULA 2	4.00	70	80	90
Ada83	4.50	60	71	80
LISP	5.00	25	64	80
FORTH	5.00	27	64	85
QUICK BASIC	5.50	38	58	90
C++	6.00	30	53	125
Ada 9X	6.50	28	49	110
Database	8.00	25	40	75
Visual Basic (Windows)	10.00	20	32	37
APL (default value)	10.00	10	32	45
SMALLTALK	15.00	15	21	40
Generators	20.00	10	16	20
Screen painters	20.00	8	16	30
SQL	27.00	7	12	15
Spreadsheets	50.00	3	6	9

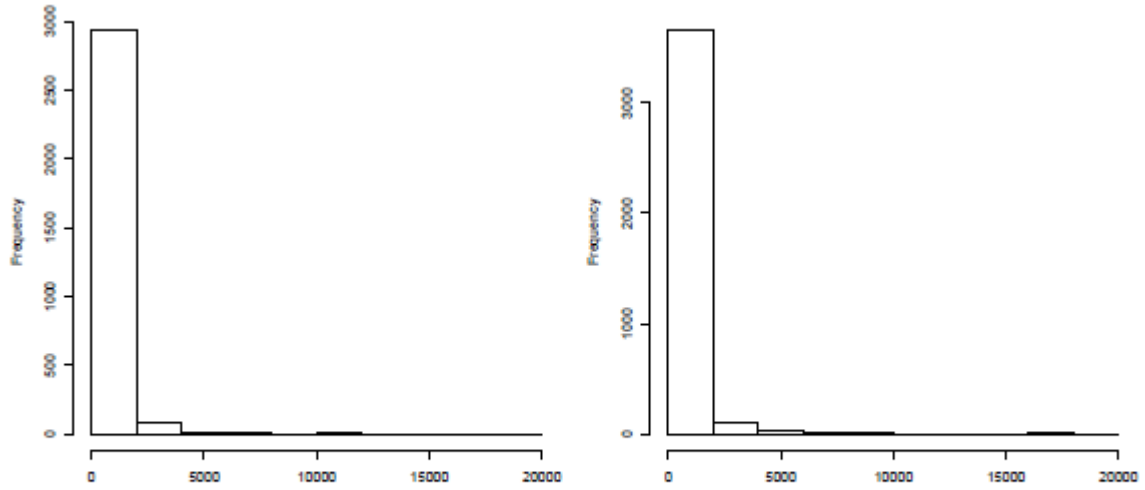
Function points: Further results and open questions

- **Further results**
  - **OO-languages**

- **Open questions**
  - **Formal study of correlation between backfiring FP and "true" FP**
  - **AOP**
  - **Evolution of functional size using FP**

How does my system compare to industrial practice?

- **ISBSG (International Software Benchmarking Standards Group)**
  - **17 countries**
  - **Release 11: > 5000 projects**
  - **Per project:**
    - **FP count, actual effort, development technologies**



(a) Unadjusted function points (b) Adjusted function points

Alternative ways of measuring the amount of functionality

- **FP: input, output, inquiry, external files, internal files**

Interface

- **Amount of functionality = size of the API**
  - **Linux kernel = number of system calls + number of configuration options that can modify their behaviour**
    - **E.g., `open` with `O_APPEND`**
    - **`ls` has 53 options + aliases** (-a and --all)

Amount of functionality in the Linux kernel

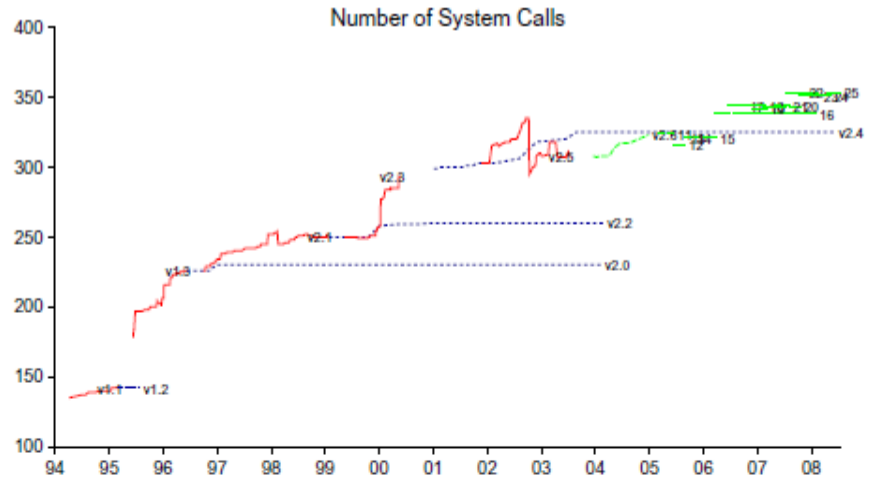


Fig. 1. The growing number of system calls and configuration options in Linux.

- **Multiple versions and variants**
  - **Production (blue dashed)**
  - **Development (red)**
  - **Current 2.6 (green)**
- **System calls: mostly added at the development versions**
  - **Rate is slowing down from 2003 – maturity?**
- **Configuration options: superlinear growth**
  - **2.5.45 – change in option format/organization**

Israeli, Feitelson

Conclusions

- **Effort and functionality estimation metrics**
  - **COCOMO, Function points…**
  - **Size of API**

2IMP25 Software Evolution

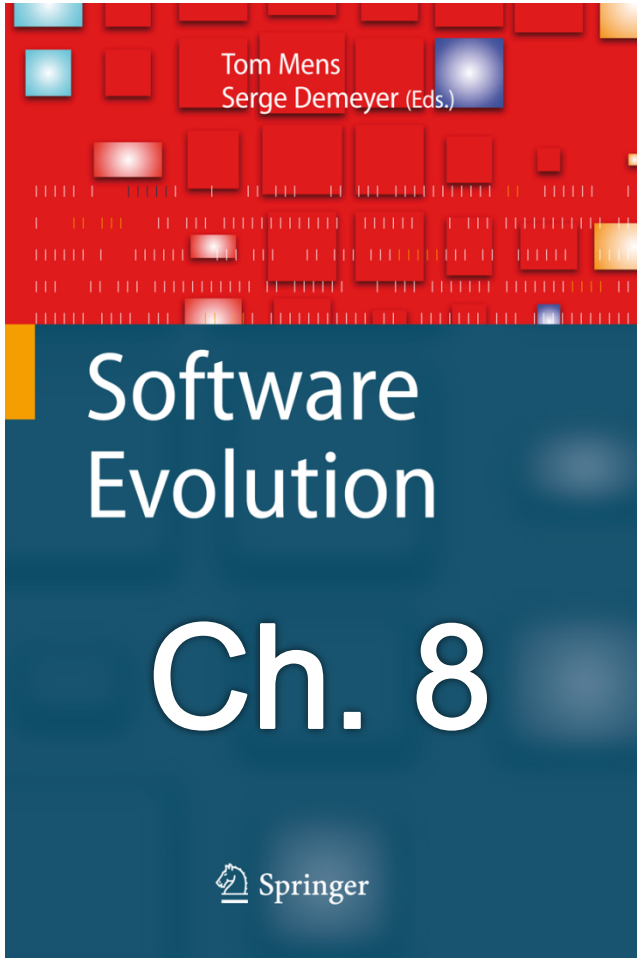
Tests

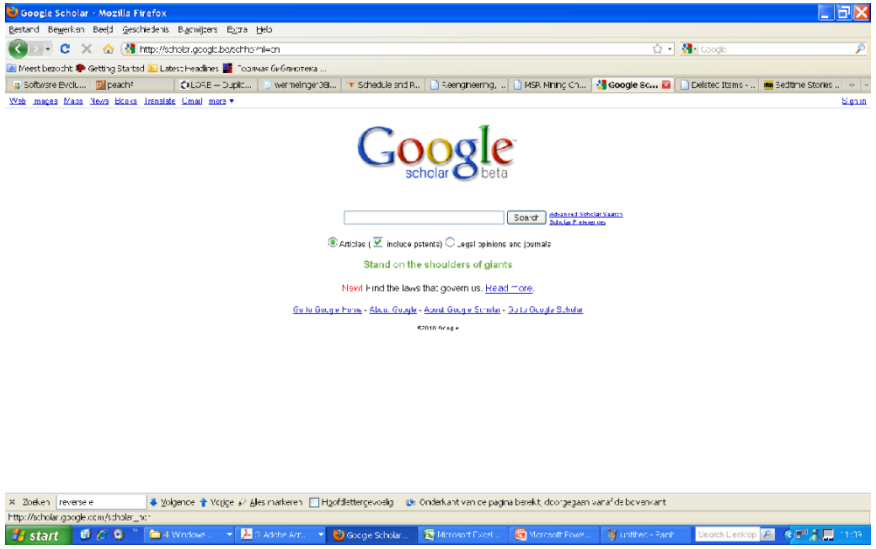
Alexander Serebrenik

Technische Universiteit
Eindhoven
University of Technology

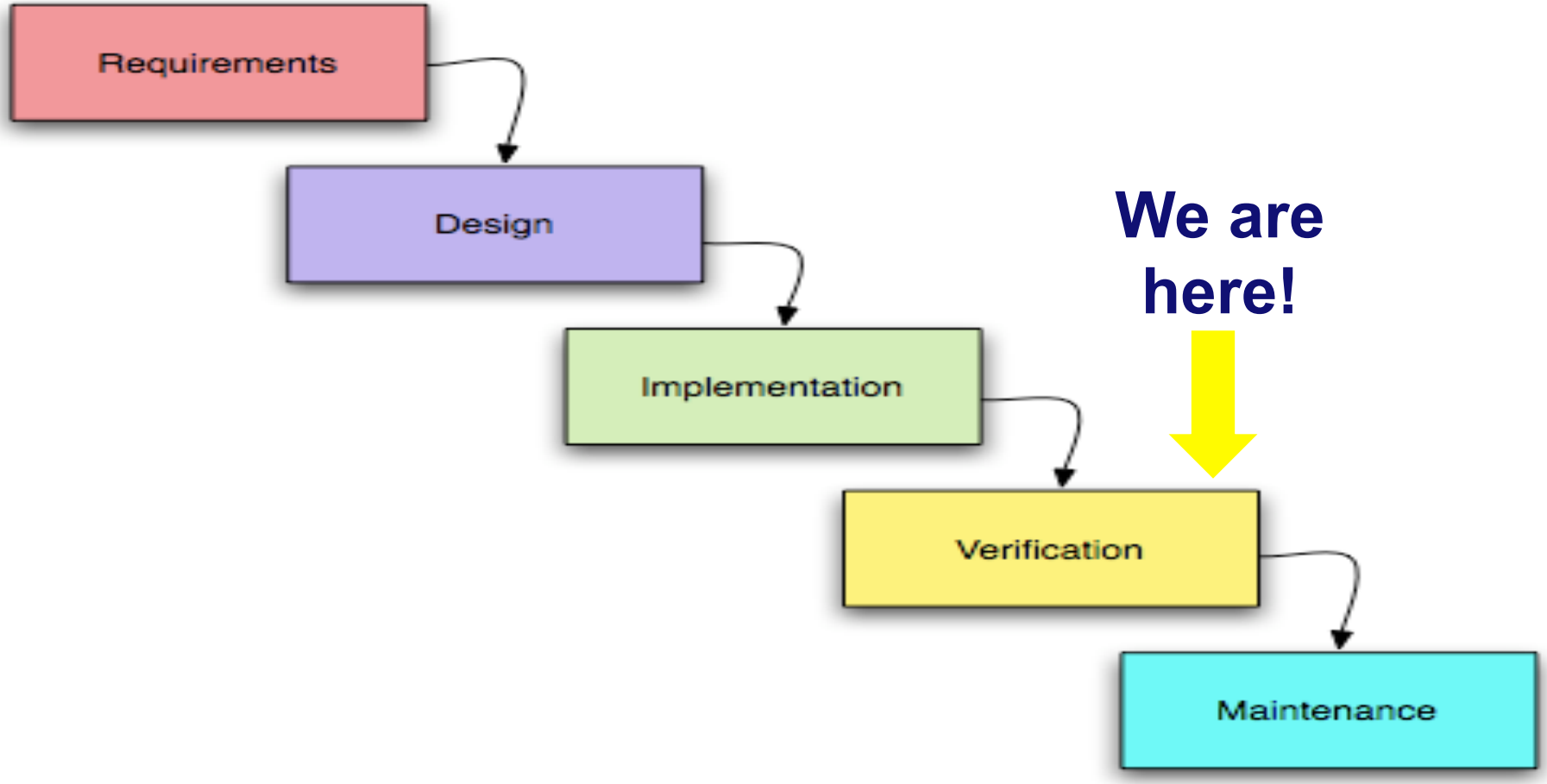
Where innovation starts

Sources


Tom Mens
Serge Demeyer (Eds.)
Software Evolution
Ch. 8
Springer


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

Waterfall model [Royce 1970]



Establishing correctness of the program

- **Formal verification**
  - **Model checking, theorem proving, program analysis**
  - **Additional artefacts: properties to be established**
  - **Optional artefacts: models**

- **Testing**
  - **Additional artefacts: test cases/scripts/programs**
  - **Optional artefacts: drivers/stubs**

- **Co-evolution problem: additional (and optional) artefacts should co-evolve with the production code**

Different flavours of tests

Testing	Kind of software		
	Management IS	Systems software	Outsourced projects
Unit	10	10	8.5
Integration	5	5	5
System	7	5	5
Acceptance	5	2.5	3

- Effort percentage (staff months) [Capers Jones 2008]
- Evolution research so far focused on unit testing
  - Highest percentage in testing
  - Best-suited for automation

Unit testing

- Test code is also code
  - Recent: unit testing frameworks become popular
- For JUnit code
  - Fixture: common part for multiple tests
    - @Before: set-up, resource claim
    - @After: resource release
  - @Test

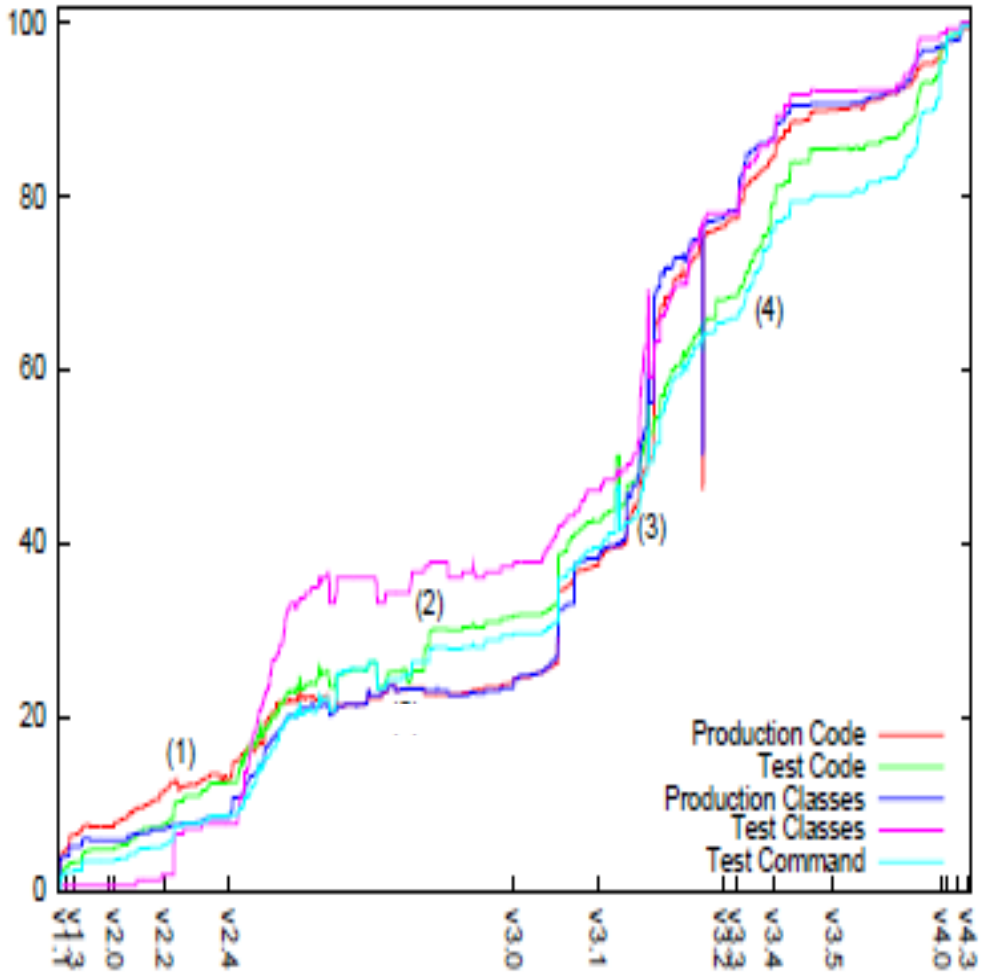
- Traditional metrics can be computed
- Compare the evolution of the production code metrics and test code metrics

Examples of co-evolution scenarios [Zaidman et al. 2008]

pLOC	tLOC	pClasses	tClasses	tCommands	interpretation
↗	→				pure development
→	↗				pure testing
↗	↗				co-evolution
→	↗	→	→		test refinement
→	→	↗	↗		skeleton co-evolution
	→		↗		test case skeletons
	→			↗	test command skeletons
→	↘				test refactoring

- p – production code
- t – testing code
- Commands – methods with @Test annotation

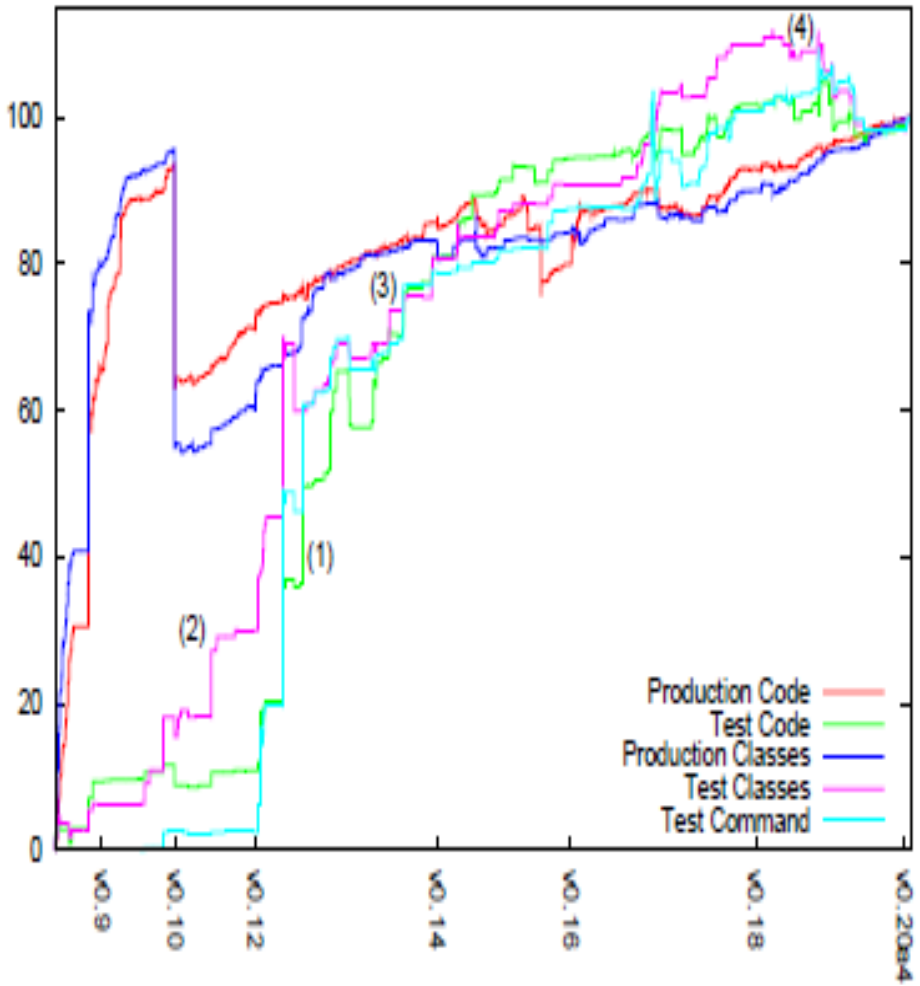
Co-evolution patterns in Checkstyle



Checkstyle, %of maximum

1. Test reinforcement: ↑ #test classes
2. Test refinement
3. Intensive development – testing backlog
4. Back to synchronous testing

Co-evolution patterns in ArgoUML



Argo UML

1. No correspondence between the production code and the test code: pure testing phase
2. Test skeleton introduction
3. Test refinement
4. Test refactorings

"Initial hill" – changes in the VCS leading to code duplication

The diagrams seem to suggest

- **Correlation between the size of the test suite size and the production code size**
  - **Reminder: McCabe's complexity is related to the expected testing effort**
  - **We are looking at the actual testing effort…**
  - **JUnit - correspondence between production and test classes**

r_s	dLOCC	dNOTC
DIT	-.0456	-.201
FOUT	.465	.307
LCOM	.437	.382
LOCC	.500	.325
NOC	.0537	-.0262
NOF	.455	.294
NOM	.532	.369
RFC	.526	.341
WMC	.531	.348

- **System: Ant**
- **Dependent variables**
  - **dLOCC – LOC per test class**
  - **dNOTC – number of test cases**
- **Independent variables**
  - **FOUT – Class-out**
  - **WMC – WMC/McCabe**
  - **LCOM – LCOM/Henderson-Sellers**

Bruntink, Van Deursen, 2004

Quantity vs. Quality

- So far: Quantity (tLOC, tClasses, tCommands)
  - BUT how good are the tests?

- Coverage: measure of test quality
  - % program components “touched” by the tests
  - Variants
    - Line coverage
    - Statement coverage
    - Function/method coverage
    - Module/class coverage
    - Block coverage
      - Block: sequence of statements with no jumps or jumps’ targets

EMMA, Open-source Java coverage tool

EMMA Coverage Report (generated Tue May 18 22:13:27 CDT 2004)

[all classes]

COVERAGE SUMMARY FOR PACKAGE [org.apache.velocity.anakia]

name	class, %	method, %	block, %	line, %
org.apache.velocity.anakia	91% (10/11)	42% (35/83)	43% (588/1382)	44% (138.9/319)

COVERAGE BREAKDOWN BY SOURCE FILE

name	class, %	method, %	block, %	line, %
Escape.java	100% (1/1)	50% (1/2)	4% (3/73)	9% (2/23)
TreeWalker.java	100% (1/1)	33% (1/3)	8% (3/38)	20% (2/10)
NodeList.java	67% (2/3)	21% (8/39)	11% (46/425)	12% (12/99)
OutputWrapper.java	100% (1/1)	50% (1/2)	16% (3/19)	25% (2/8)
AnakiaJDOMFactory.java	100% (1/1)	40% (2/5)	30% (8/27)	50% (3/6)
XPathTool.java	100% (1/1)	50% (2/4)	40% (12/30)	60% (3/5)
AnakiaElement.java	100% (1/1)	50% (6/12)	51% (37/73)	44% (7/16)
AnakiaTask.java	100% (1/1)	92% (12/13)	68% (443/656)	71% (99.5/141)
XPathCache.java	100% (1/1)	67% (2/3)	80% (33/41)	76% (8.4/11)

[all classes]

EMMA 2.0.4015 (stable) (C) Vladimir Roubtsov

What happens if a line is covered only partially?

- **EMMA:**

```
 1 public class MyClass
 2 {
 3     public static void main (final String [] args)
 4     {
 5         int vi = 1;
 6         int vj = vi > 0 ? -1 : 1;
 7
 8         for (int vk = 0; vk < vj; ++ vk)
 9         {
10             System.out.println ("vk = " + vk);
11         }
12     }
13
14     public MyClass () {}
15 }
```

- **Which parts of the yellow lines are covered and which parts are not?**

Condition coverage vs. Decision coverage

- **Condition coverage**
  - **Every boolean subexpression has been evaluated to true and to false**
- **Decision coverage**
  - **In every decision (if/loop) both the true and the false branch have been tested**
- **Does condition coverage imply decision coverage?**
- **Does decision coverage imply condition coverage?**

Condition coverage vs. decision coverage

```
int foo(int a, int b) {
   int c = b;

   if ((a>5) && (b>0)) {
      c = a;
   }

   return a*c;
}
```

- { foo(7,-1), foo(4,2) } covers all conditions but not all decisions
  (T,F) and (F,T)
- { foo(7,-1), foo(7,1) } covers all decisions but not all conditions
  (T,F) and (T,T)

Condition coverage vs. decision coverage

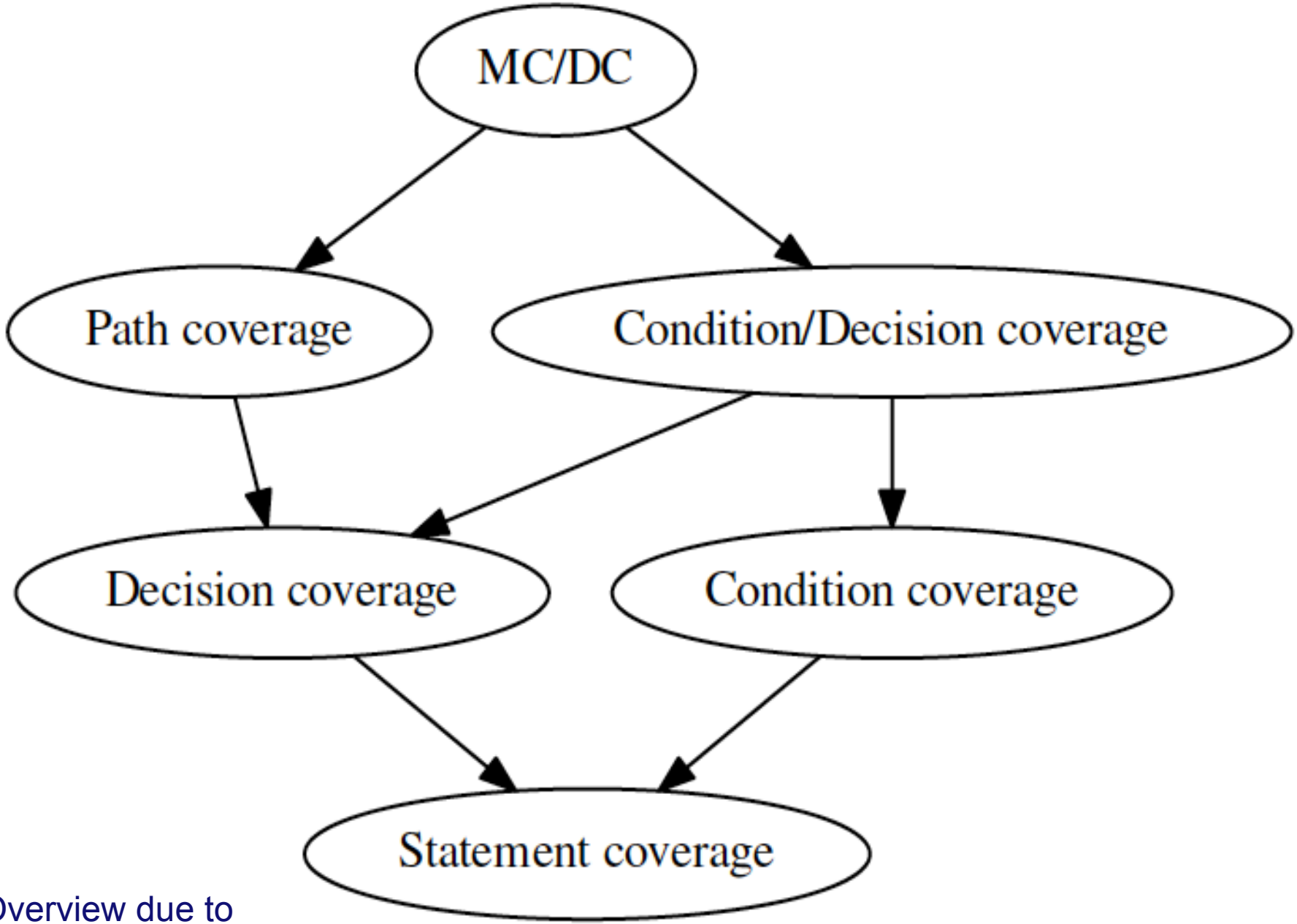
```
int foo(int a, int b) {
   int c = b;

   if ((a>5) && (b>0)) {
      c = a;
   }

   return a*c;
}
```

- { foo(7,-1), foo(4,2) } covers all conditions but not all decisions
  (T,F) and (F,T)
- { foo(7,-1), foo(7,1) } covers all decisions but not all conditions
  (T,F) and (T,T)

- Condition/decision coverage: both condition coverage and decision coverage
  - { foo(7,-1), foo(4,2), foo(7,1) }

Path coverage

- **Path coverage: all possible paths through the given program**
  - **Unrealistic: n decisions ⇒ up to 2^n different paths**
  - **Some paths are infeasible**
    - **Whether a path is infeasible is undecidable**

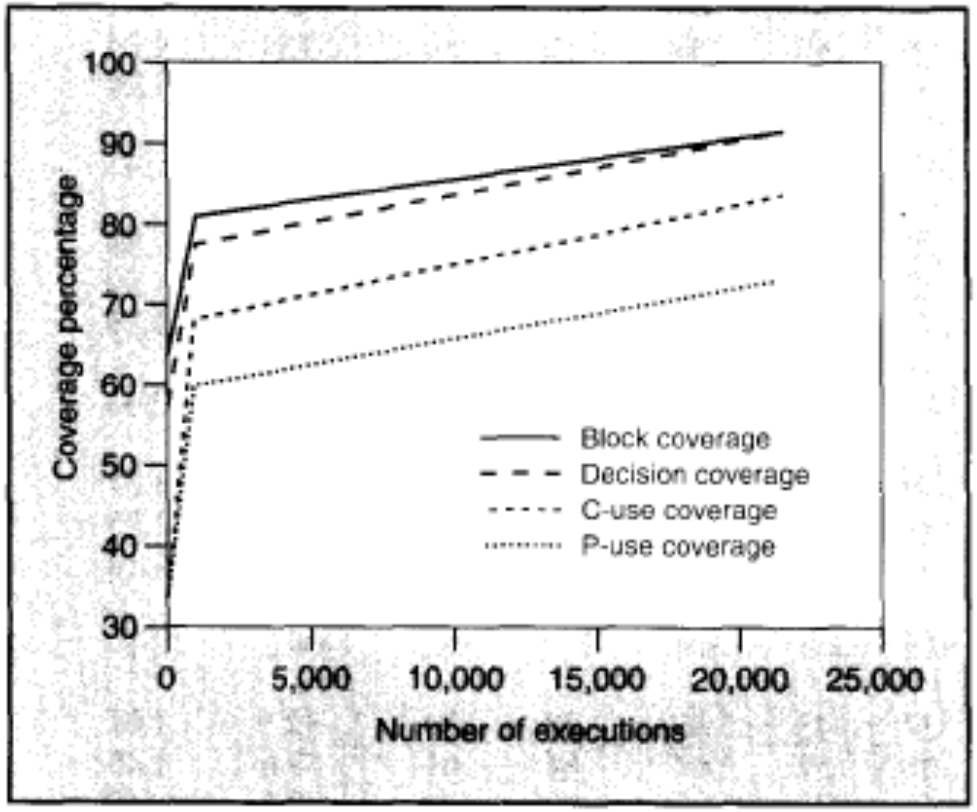


Overview due to [Steneker 2016]

Not all paths are equally important

- **Special paths: from definition (i = 1) to use (x += i)**
  - **c-use if the use is a computation (x += i)**
  - **p-use if the use is a predicate (x < i)**

The more you test the better the coverage



Horgan, London, Lyu

- Average over 12 competing versions of the same software
- Coverage increases
  - 100% is still a dream even after more than 20,000 tests!

Popular test coverage tools (Steneker, 2016)

Tool	Statement	Function	Decision (branch)	Condition	Condition/Decision	MC/DC	Path
BullseyeCoverage (Bullseye) [94]		✓	✓	✓	✓		
Squish Coco (FrogLogic) [95]	✓		✓	✓			
CTC (Testwell) [96]	✓	✓	✓	✓	✓	✓	
gcov/lcov (SourceForge) [97]	✓	✓	✓				
PureCoverage (IBM) [98]	✓	✓					
C++Test (Parasoft) [99]	✓		✓	✓		✓	✓
VectorCAST (Vector Software) [100]	✓		✓			✓	
NCover (NCover) [101]	✓		✓	✓			
OpenCover (open source) [102]	✓	✓	✓				
Jacoco (open source) [103]	✓	✓	✓				
Cobertura (open source) [104]	✓		✓				

Christine Gerpheide (MSc, 2014, collab ASML)

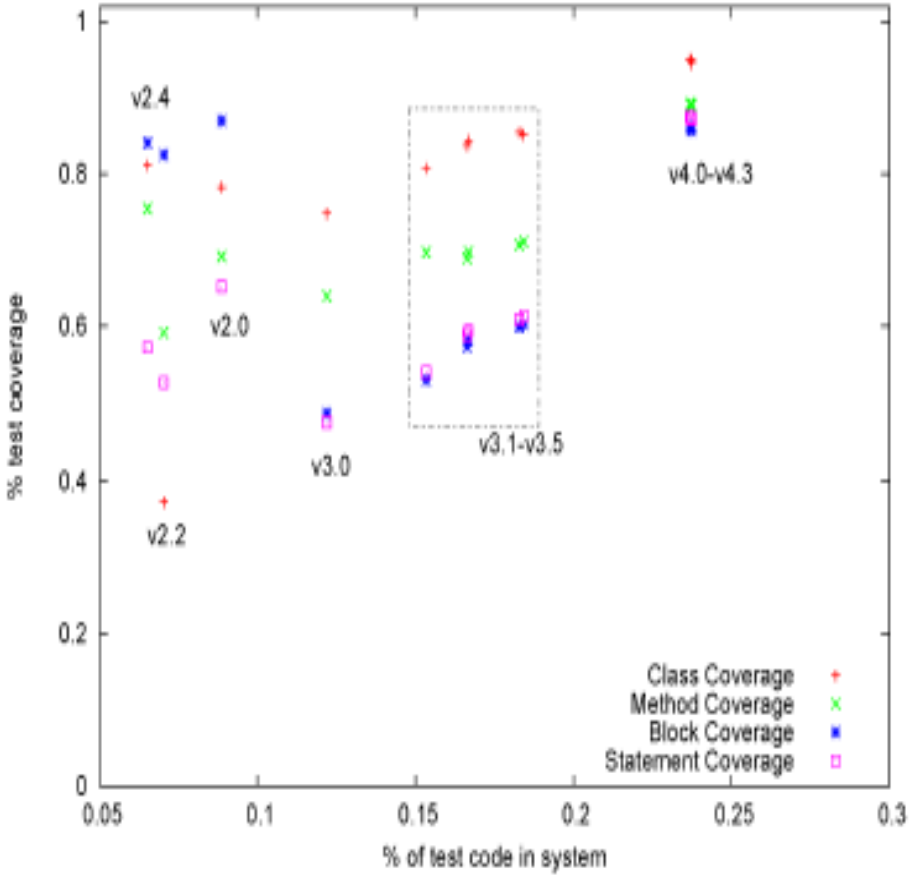
HelloWorld.qvto | AnotherTrans.qvto | ddf2basics.qvto | ddf2pgwb.qvto | pgwb_helpers.qvto

```
import AnotherTrans;

modeltype ABC uses ABC('http:///ABC.ecore');

transformation HelloWorld(in source:ABC, out target:ABC)
access transformation AnotherTrans();

main() {
    source.rootObjects()[Root]->map Root2Root();
}

mapping Root :: Root2Root() : Root {
    element += self.allSubobjects()[A]->map A2B();
}

mapping A :: A2B() : B when { self.id > 0 } {
    if (self.a = "hello") then { b := "world"}
    else {b := "bye"}
    endif;

}

constructor B::B(x : A) {
    b := x.a;
```

Problems | Javadoc | Declaration | Console | Progress | Action Items | Coverage View

Module	Total function coverage	Mapping Coverage	Helper Coverage	Constructor Coverage	Estimated Expression Coverage
MyTester	35% (5 of 14)	36% (4 of 11)	50% (1 of 2)	0% (0 of 1)	60% (57 of 95)
AnotherTrans	27% (3 of 11)	22% (2 of 9)	50% (1 of 2)	100% (0 of 0)	51% (34 of 66)
HelloWorld	66% (2 of 3)	100% (2 of 2)	100% (0 of 0)	0% (0 of 1)	80% (24 of 30)

What about evolution of test coverage?

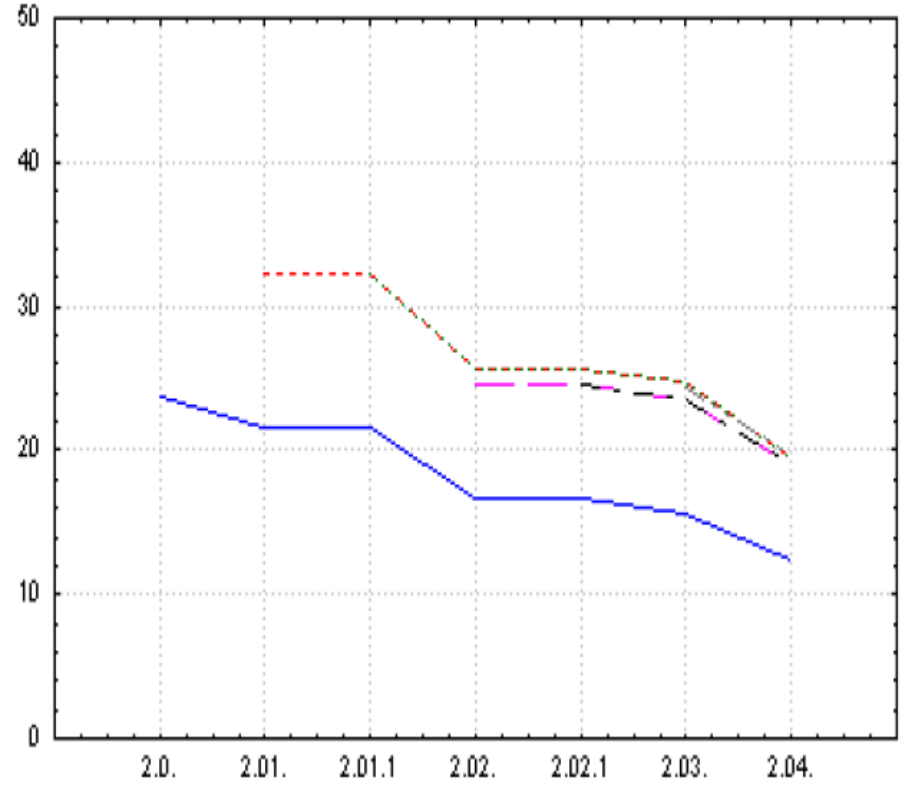


Checkstyle
Abscisse tLOC/(tLOC+pLOC)

[Zaidman et al. 2008]

- **High class coverage (>80% and >95% for 4.*)**
  - **Exception: 2.2**
- **2.***
  - **⇐: pLOC increases faster than tLOC**
  - **drop in coverage values: major reengineering**
- **3.0-4.0: increase for all forms of coverage**

Function coverage in bash

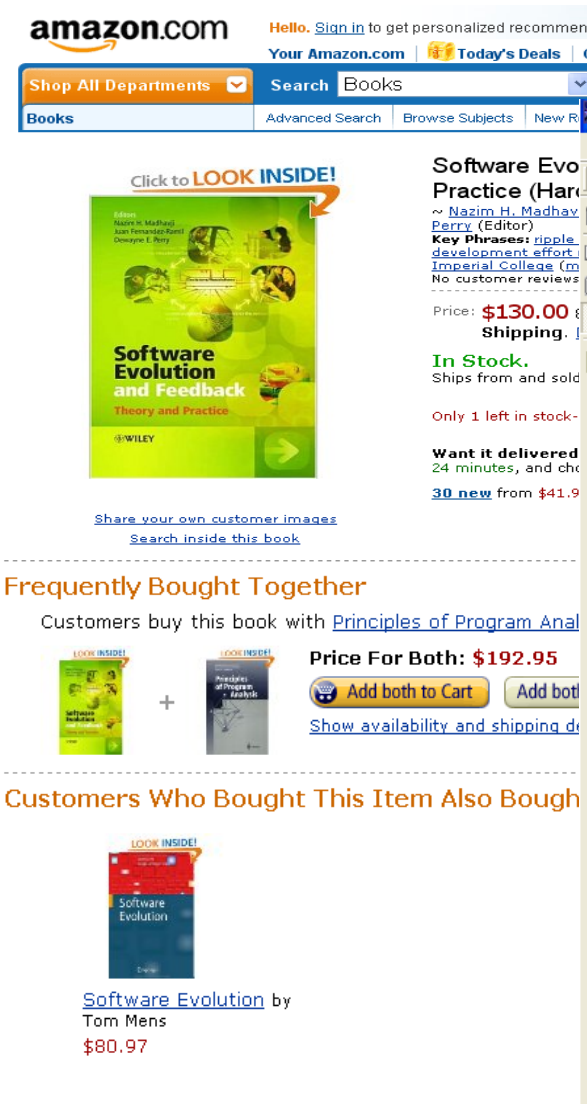


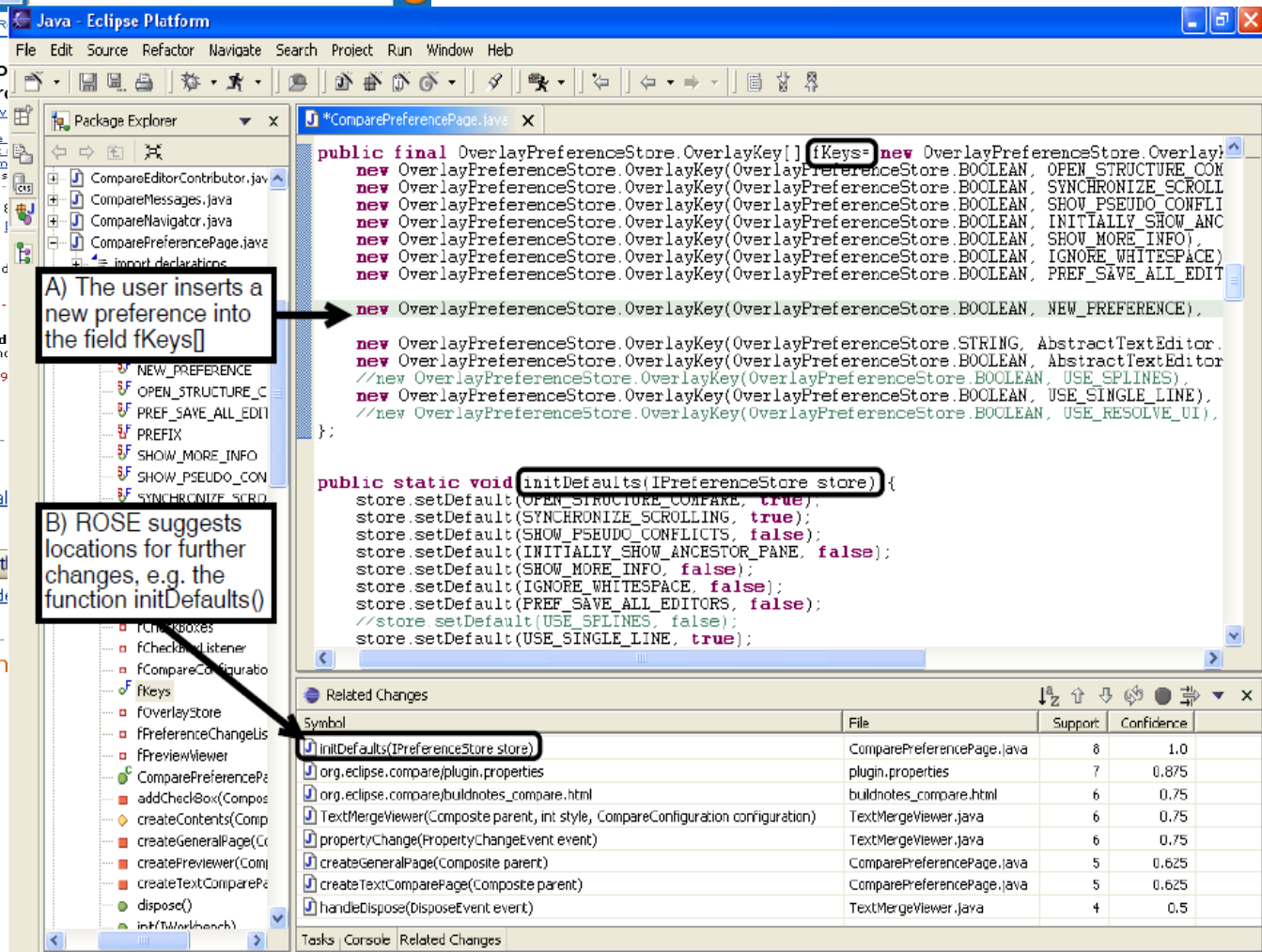
Bash
Elbaum, Gable, Rothermel

- **Retrospective analysis: tests for version i were rerun for all versions j, j>i**
- **Function coverage**
- **BUT #functions increases and coverage is percentage**
  - **Consider only functions present in all Bash versions**

Closer look at changes

- **Remember eROSE? [Zimmermann et al. 2004]**





Association Rule Mining

- eROSE is based on detecting frequent sets and association rules, i.e., elements that often are changed together
  - Popular technique: Apriori algorithm

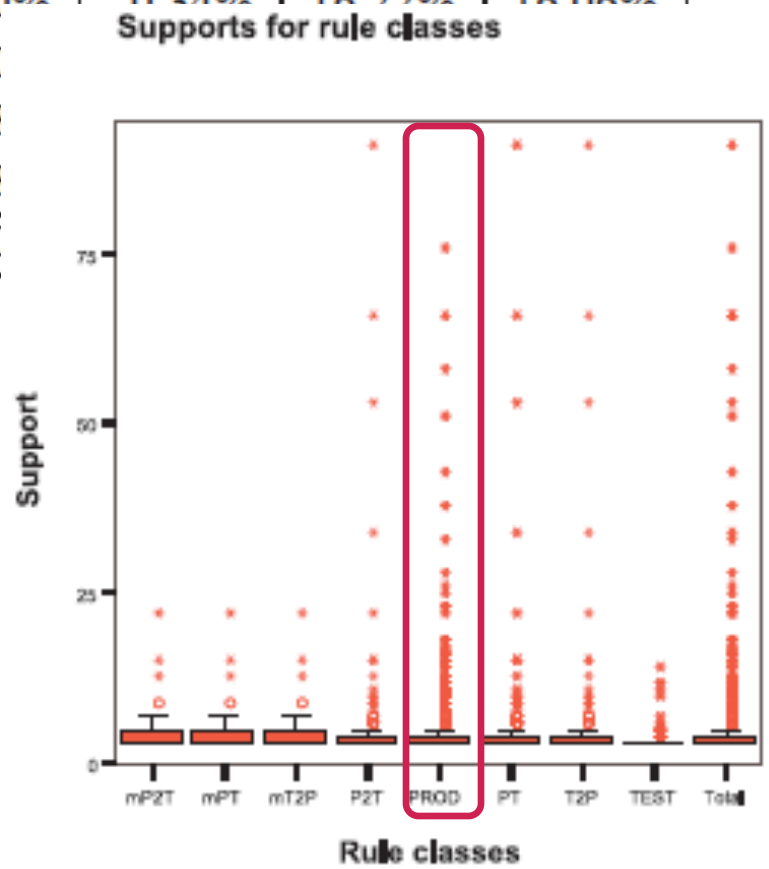
- Tests are code, so [Lubsen, M.Sc. thesis]
  - Distinguish tests/production classes based on their names
    - Drop files that are neither source nor test (makefiles, images, etc.)
  - Use Apriori to mine association rules

Rule categorization

- Categorize rules A⇒B (A, B – classes):
  - PROD: A and B are production classes
  - TEST: A and B are test classes
  - P&T pairs:
    - P2T, T2P
    - mP2T, mT2P: matched pairs {*C.java* ⇒ *CTest.java*}
- Are there any other types of rules we've missed?

Empirical evaluation

Rule Class	Checkstyle	A.I	A.II	B.I	B.II	C.I	C.II
ALL (N)	58566	101896	14590	8820	219248	27308	498
PROD	98,86%	35,15%	49,64%	39,00%	99,12%	51,84%	40,96%
TEST	0,48%	26,11%	9,95%	24,81%	0,20%	9,99%	16,87%
P&T	0,67%	38,75%	40,25%	36,19%	0,69%	32,44%	32,13%
P2T	0,33%	19,37%	20,12%	18,10%	0,34%	16,22%	16,06%
T2P	0,33%	19,37%	20,12%	18,1(			
mP2T	0,09%	0,78%	0,74%	0,83			
mT2P	0,09%	0,78%	0,74%	0,83			
UNDEF	0,00%	0,00%	0,16%	0,0(			



- **Checkstyle:**
  - **Large number of commits with many production classes**
    - **Classes are together by chance**
    - **Support is very low**
  - **Commits on test classes involve only few of them**

Quality of rules: A ⇒ B (A, B – sets)

- **Support** $|A \wedge B| = P(A,B)$
- **Confidence** $|A \wedge B| : |A| = P(B|A)$

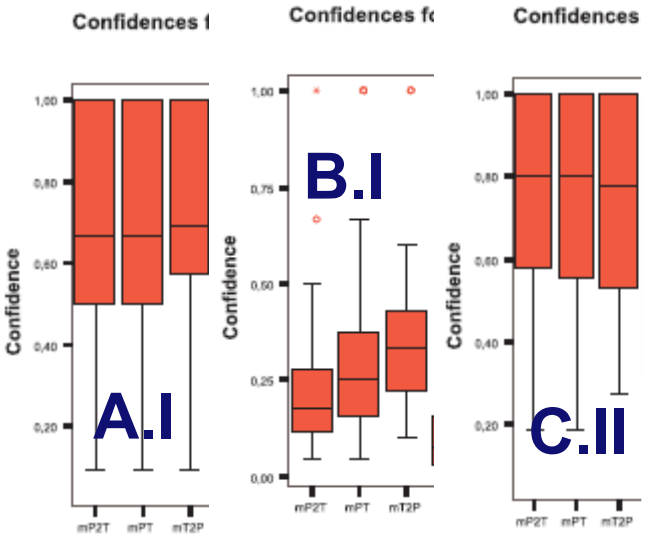
- **Strong rule: high confidence and reasonable support**

- **There are more ways to assess quality of the rules!**

Empirical evaluation

Rule Class	Checkstyle	A.I	A.II	B.I	B.II	C.I	C.II
ALL (N)	58566	101896	14590	8820	219248	27308	498
PROD	98,86%	35,15%	49,64%	39,00%	99,12%	51,84%	40,96%
TEST	0,48%	26,11%	9,95%	24,81%	0,20%	9,99%	16,87%
P&T	0,67%	38,75%	40,25%	36,19%	0,69%	32,44%	32,13%
P2T	0,33%	19,37%	20,12%	18,10%	0,34%	16,22%	16,06%
T2P	0,33%	19,37%	20,12%	18,10%	0,34%	16,22%	16,06%
mP2T	0,09%	0,78%	0,74%	0,83%	0,01%	0,78%	4,82%
mT2P	0,09%	0,78%	0,74%	0,83%	0,01%	0,78%	4,82%
UNDEF	0,00%	0,00%	0,16%	0,00%	0,00%	5,73%	10,04%

- **A.I, A.II, C.I and C.II (synchronous co-evolution)**
  - **the ratios correspond to the effort distribution.**
  - **the confidence of typical rules is not low.**



Question

- Apriori algorithm usually works for A and B as sets of elements rather than individual elements:
  - Age > 52, CurrentAcc = true ⇒ Income > 43759, SavingsAcc = true

- Why did Lubsen consider only pairs of classes?

More than JUnit

- There exist JUnit-like systems for
  - Server-side code: Cactus
    http://jakarta.apache.org/cactus/
  - Web-applications: HttpUnit
    http://sourceforge.net/projects/httpunit/
  - Popularity?
  - No research so far (AFAIK)

Conclusions

- Verification ⇒ Testing ⇒ Unit testing
  - Dr. Anton Wijs: incremental model checking

- Unit testing – another group of code files
  - Traditional metrics are applicable
    - Correlation, co-evolution patterns
  - Coverage metrics
  - Association rules