2IS55 Software Evolution

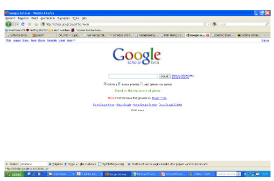
Software metrics

Alexander Serebrenik

Assignments

- Assignment 5:
  - Deadline: April 26
  - “Cathedral” / “bazaar”

Sources



Recap: Version control systems

- Version control systems
  - Centralized vs. distributed
  - File versioning (CVS) vs. product versioning
  - Logs can be used to get insights in
    - How humans work?
    - How do the files evolve?

Today: Version control system is not just a log…

r1108668 | tokoe | 2010-03-29 16:54:02 +0200 (ma, 29 mrt 2010)
Changed paths:
M /trunk/KDE/kdepim/kmail/kmsearchpatternedit.cpp

Revision 1108667

Revision 1108668

- Measure each revision
- Get insights in the evolution

Why do we want to measure revisions?

- Recall the “goals-questions-<views>-metrics” approach we used for architecture reconstruction?
  - Goals: What problem does the measurement try to solve?
    - Ex.: Modifying code is experienced as difficult
    - Goal: Assess and improve maintainability of the code
  - Questions: What do we need to know to achieve the goal?
    - Is the code large? Complex? Appropriately modularized? Buggy? Documented?
  - <Views>: Which views are need to answer the questions?
    - Individual components, dependency structure
  - Metrics: How can we quantify the answers?
    - Main topic of the lecture

Measure each revision…

- **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**

Domains and scales

Nominal
=,≠ 1-1 trans. Imple-mentation language

Domains and scales

Nominal	Ordinal
=,≠ 1-1 trans. Imple-mentation language	> >-pres. trans. Priorities (high > middle > low)

Domains and scales

Nominal	Ordinal	Interval
=,≠ 1-1 trans. Imple-mentation language	> >-pres. trans. Priorities (high > middle > low)	Distance function Affine Tempe-rature (°C, °F)

Domains and scales

Nominal	Ordinal	Interval	Ratio
=,≠ 1-1 trans. Imple-mentation language	> >-pres. trans. Priorities (high > middle > low)	Distance function Affine Tempe-rature (°C, °F)	Zero, unit Linear m ↔ ft % of commits by Alice

Domains and scales

Nominal	Ordinal	Interval	Ratio	Absolute

=,≠
1-1 trans.
Imple-mentation language
	>
	>-pres. trans.
	Priorities (high > middle > low)
		Distance function
		Affine
		Tempe-rature (°C, °F)
			Zero, unit
			Linear
			m ↔ ft
			% of commits by Alice
				Values are absolute
				Identity
				#devel’s
				P(failure)

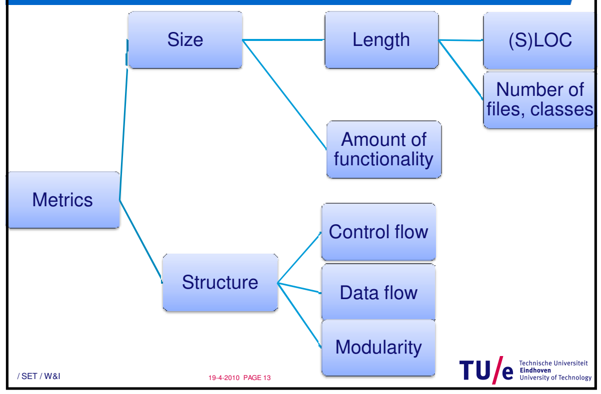
Metrics and scales

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

To what scale does it belong?

Nominal	Ordinal	Interval	Ratio	Absolute
Imple-mentation language	Priorities (high > middle > low)	Tempe-rature (°C, °F)	m ↔ ft	#developers

Classification of metrics [à la Fenton, Pfleeger 1996]



Metrics: how do we discuss them?

- Definition and variants
- Advantages and disadvantages
- Metrics distribution studies
- Evolutionary studies

Related to Lehman's 2:
• As an E-type system is changed its complexity increases <…> unless work is done to maintain or reduce the complexity.

Program length (LOC)

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

```
for (i = 0;
       i < 100;
          i += 1) {
     printf("hello");
}

/* 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

Advantages of (S)LOC: Effort estimation

- Used for software engineering project management
- 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

$$E = aS^b \qquad T = cE^d$$

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

	a	b	c	d
Small experienced team, flexible requirements	2.4	1.05	2.5	0.38
Tight requirements	3.6	1.20	2.5	0.32

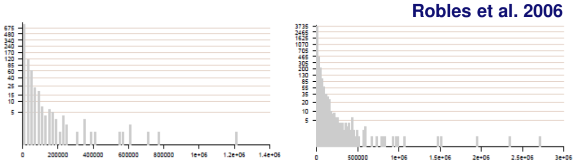
More advanced COCOMO: even more categories

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

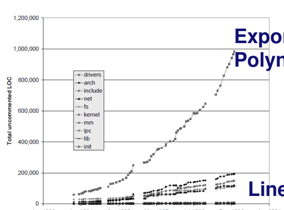


- Distribution of SLOC in Debian 2.0 (left) and 3.0 (right)
- Controversy: log-normal or double Pareto?
  - Importance: knowing distribution one can estimate the probability to obtain files of a given size
  - Hence, to estimate size of the entire system
  - And the effort required (COCOMO)

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?

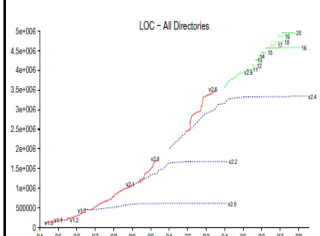


Exponential?
Polynomial?

Linear

- Godfrey and Tu: superlinear growth is typical for OS
  - Does this remind you of Cathedral and Bazaar?
  - 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

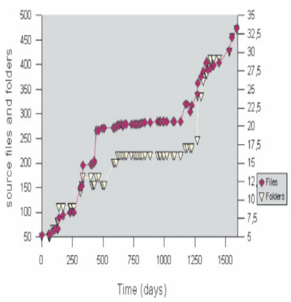
- Different variants: LOC, SLOC, LLOC
- Advantages:
  - Easy to compute, moderately correlates with #bugs
  - Can be used to estimate the development effort (COCOMO)
- Disadvantages
  - Different programming languages and developers
  - Hand-written vs. generated code
- Distribution "exponential-like"
- 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)

#components: Advantages

- Intuitive and easy to calculate
- 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!



Growth in Gaim (Ramil, Capiluppi 2004)

However…

- The Lehman's law talks about functional capability
- How can we measure amount of functionality in the system?
  - [Albrecht 1979] "Function points"
    - Anno 2010: Different variants: IFPUG, NESMA, …
  - Determined based on system description
    - \+ Independent from the specific implementation language or technology
    - – Requires knowledge of a certified expert
  - Amount of functionality can be used to assess the development effort and time before the system is built
  - Originally designed for information systems

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

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

Alternative ways of measuring the amount of functionality

- FP: $\underbrace{\text{input, output, inquiry, external files}}_{\text{Interface}}$, internal files

- 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

Amount of functionality in the Linux kernel

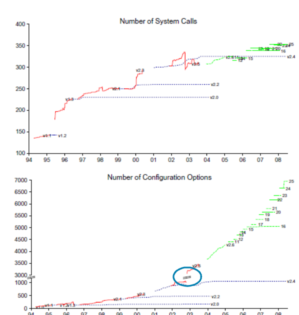
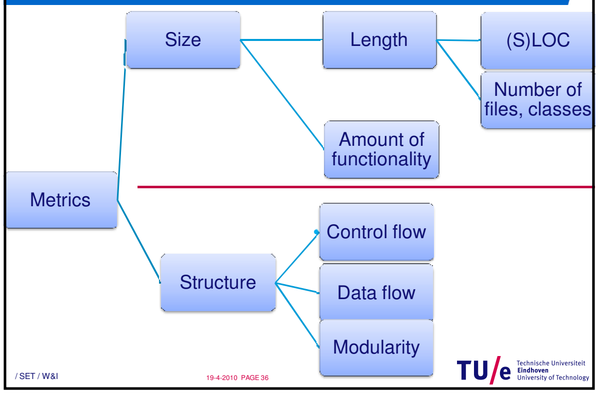


Fig. 1. The growing number of system calls and configuration options in Linux.

- 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

So far



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 ≤ Volume(function) ≤ 1000
    - 100 ≤ 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

```
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 $\log_2(24)$ ≈ 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)

Conclusions

- Software metrics are used to assess maintainability and evolution
- Size
  - LOC, SLOC, LLOC
  - Amount of functionality (FP, size of the API)
- Complexity
  - Halstead: volume V, effort E, time T and #bugs B
- Next time
  - McCabe's cyclomatic complexity: v(G)
  - Combined metrics: Maintainability index
  - OO and more!