Software metrics

Alexander Serebrenik

Assignments

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

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...

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
• Get insights in the evolution
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
What metrics have we seen so far?
- Size: LOC, SLOC
- Code duplication: POP
- Requirements: Flesch-Kincaid grade level

To what scale does it belong?
- Nominal: Implementation language
- Ordinal: Priorities (high > middle > low)
- Interval: Temperature (°C, °F)
- Ratio: m ft
- Absolute: #developers

Classification of metrics [a la Fenton, Pfleeger 1996]
- Size
- Length
- (S)LLOC
- Amount of functionality
- Number of files, classes
- Structure
- Control flow
- Data flow
- Modularity

What happens with tables or reduce the complexity.

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

Correction coefficients come from
- 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

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

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

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

\[ E = aS^b \quad T = cE^d \]

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

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

- 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

LOC in Linux kernel

- 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 3.1. The Initial IFPUG Version of the Function Point Metric

<table>
<thead>
<tr>
<th>Significant Parameter</th>
<th>Low Complexity</th>
<th>Medium Complexity</th>
<th>High Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>External input</td>
<td>x ≤ 5</td>
<td>x ≤ 4</td>
<td>x ≤ 4</td>
</tr>
<tr>
<td>External output</td>
<td>x ≤ 4</td>
<td>x ≤ 5</td>
<td>x ≤ 7</td>
</tr>
<tr>
<td>Logical interface</td>
<td>x ≤ 5</td>
<td>x ≤ 10</td>
<td>x &gt; 10</td>
</tr>
<tr>
<td>External interface</td>
<td>x ≤ 5</td>
<td>x ≤ 7</td>
<td>x &gt; 10</td>
</tr>
<tr>
<td>External inquiry</td>
<td>x ≤ 5</td>
<td>x ≤ 4</td>
<td>x ≤ 6</td>
</tr>
</tbody>
</table>

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

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: 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

Amount of functionality in the Linux kernel

- 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

- Size
- Length
- Number of files, classes
- Amount of functionality
- Control flow
- Data flow
- Modularity

Metrics

Halstead metrics

- Four basic metrics of Halstead

<table>
<thead>
<tr>
<th></th>
<th>Total</th>
<th>Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operators</td>
<td>N1</td>
<td>n1</td>
</tr>
<tr>
<td>Operands</td>
<td>N2</td>
<td>n2</td>
</tr>
</tbody>
</table>

- Length: \( N = N_1 + N_2 \)
- Vocabulary: \( n = n_1 + n_2 \)
- Volume: \( V = N \log_2 n \)
- Insensitive to lay-out
- VeritySoft:
  - \( 20 \leq \text{Volume(function)} \leq 1000 \)
  - \( 100 \leq \text{Volume(file)} \leq 8000 \)

Complexity metrics: Halstead (1977)

- Sometimes is classified as size rather than complexity
- Unit of measurement
  - Parts of a statement:
    - LOC, SLOC, LLOC, classes and packages
  - Operators:
    - Traditional (+, -, >, keywords (return, if, continue)
    - Operators:
      - Identifiers, constants

Halstead metrics: Example

```c
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

<table>
<thead>
<tr>
<th></th>
<th>Total</th>
<th>Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operators</td>
<td>N1</td>
<td>n1</td>
</tr>
<tr>
<td>Operands</td>
<td>N2</td>
<td>n2</td>
</tr>
</tbody>
</table>

Effort: \( E = V \times D \)

Further Halstead metrics

<table>
<thead>
<tr>
<th></th>
<th>Total</th>
<th>Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operators</td>
<td>N1</td>
<td>n1</td>
</tr>
<tr>
<td>Operands</td>
<td>N2</td>
<td>n2</td>
</tr>
</tbody>
</table>

- Volume: \( V = N \log_2 n \)
- Difficulty: \( D = ( n_1 / 2 ) \times ( N_2 / n_2 ) \)
- Sources of difficulty: new operators and repeated operands
- Example: \( 17/2 \times 30/7 = 36 \)
- Effort: \( E = V \times D \)
- Time to understand/implement (sec): \( T = E / 18 \)
- Running example: 793 sec = 13 min
- Does this correspond to your experience?
- Bugs delivered: \( E^{10} / 3000 \)
- 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```

<table>
<thead>
<tr>
<th></th>
<th>Total</th>
<th>Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operators</td>
<td>N1</td>
<td>n1</td>
</tr>
<tr>
<td>Operands</td>
<td>N2</td>
<td>n2</td>
</tr>
</tbody>
</table>

```
i++```

<table>
<thead>
<tr>
<th></th>
<th>Total</th>
<th>Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operators</td>
<td>N1</td>
<td>n1</td>
</tr>
<tr>
<td>Operands</td>
<td>N2</td>
<td>n2</td>
</tr>
</tbody>
</table>

- 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!