Softwaretechnik / Software-Engineering

Lecture 04: More Process Modelling & Software Metrics

2015-05-04

Prof. Dr. Andreas Podelski, Dr. Bernd Westphal

Albert-Ludwigs-Universität Freiburg, Germany
Contents & Goals

Last Lecture:
- process, model, process vs. procedure model
- code & fix, waterfall, S/P/E programs, (rapid) prototyping

This Lecture:
- **Educational Objectives:** Capabilities for following tasks/questions.
  - what is evolutionary, incremental, iterative?
  - what’s the fundamental idea of the spiral model? where’s the spiral?
  - what is the difference between procedure and process model?
  - what are the constituting elements of “V-Modell XT”? what project types does it support, what is the consequence? what is tailoring in the context of “V-Modell XT”?
  - what are examples of agile process models? what are their principles? describe XP, Scrum
  - what is a nominal, . . . , absolute scale? what are their properties?
  - which properties make a metric useful?
  - what’s the difference between objective, subjective, and pseudo metrics?
  - compute LOC, cyclomatic complexity, LCOM, . . . for this software

- **Content:**
  - non-linear procedure models cont’d, process models (V-Modell XT, Scrum, . . .)
  - scales, metrics
Non-Linear Procedure Models
Evolutionary and Iterative Development

**Evolutionary and Iterative Development**

---

**Evolutionary software development** — an approach which includes evolutions of the developed software under the influence of practical/field testing. New and changed requirements are considered by developing the software in **sequential steps of evolution**.

Ludewig & Lichter (2013), flw. (Züllighoven, 2005)

**Iterative software development** — software is developed in **multiple iterative steps**, all of them planned and controlled. Goal: each iterative step, beginning with the second, corrects and improves the existing system based on defects detected during usage. Each iterative steps includes the characteristic activities **analyse, design, code, test**.

Ludewig & Lichter (2013)
**Incremental Development**

*incremental software development* — The total extension of a system under development remains open; it is realised in *stages of expansion*. The first stage is the **core system**. Each stage of expansion extends the existing system and is subject to a separate project. Providing a new stage of expansion typically includes (as with iterative development) an improvement of the old components.

**Ludewig & Lichter (2013)**

- **Note**: (to maximise confusion) IEEE calls our “iterative” incremental:

*incremental development* — A software development technique in which requirements definition, design, implementation, and testing occur in an overlapping, iterative (rather than sequential) manner, resulting in incremental completion of the overall software product.

**IEEE 610.12 (1990)**

- One difference (in our definitions):
  - **iterative**: steps towards fixed goal,
  - **incremental**: goal extended for each step; next step goals may already be planned.

**Examples**: operating system releases, short time-to-market (→ continuous integration).
The Spiral Model
**Quick Excursion: Risk and Riskvalue**

**risk** — a problem, which did not occur yet, but on occurrence threatens important project goals or results. Whether it will occur, cannot be surely predicted.

Ludewig & Lichter (2013)

\[
\text{riskvalue} = p \cdot K
\]

\(p\): probability of problem occurrence, \(K\): cost in case of problem occurrence.

- **Avionics** requires: “Average Probability per Flight Hour for Catastrophic Failure Conditions of \(10^{-9}\) or ‘Extremely Improbable’” (AC 25.1309-1).
- “problems with \(p = 500 \cdot 10^{-3} = 0.5\) are not risks, but environment conditions to be dealt with”
The Spiral Model (Boehm, 1988)

Repeat until end of project (successful completion or failure):

(i) **determine** the set $R$ of **risks threatening** the project;
   if $R = \emptyset$, the project is successfully completed

(ii) **assign** each risk $r \in R$ a **risk value** $v(r)$

(iii) for the risk $r_0$ with the **highest risk value**, $r_0 = \max\{v(r) \mid r \in R\}$,
   find a way to eliminate this risk, and go this way;
   if there is no way to eliminate the risk, stop with project failure

**Advantages:**

- we know early if the project goal is unreachable,
- knowing that the biggest risks are eliminated gives a good feeling.

**Note:** **risk** can be anything; e.g. open technical questions (→ prototype?), but also
lead developer leaving the company (→ invest in documentation), changed market
situation (→ adapt appropriate features), …
A concrete process using the Spiral Model could look as follows:

- fix goals, conditions,
- risk analysis,
- develop and test,
- plan next phase,
Process Models
A process model may describe:

- organisation, responsibilities, roles;
- structure and properties of documents;
- methods to be used, e.g. to gather requirements or to check intermediate results;
- steps to be conducted during development, their sequential arrangement, their dependencies (the procedure model);
- project phases, milestones, testing criteria;
- notations and languages;
- tools to be used (in particular for project management).

Process models typically come with their own terminology (to maximise confusion?), e.g. what we call artefact is called product in V-Model terminology.

Process models are legion; we will take a closer look onto:

- V-Model XT, (Rational) Unified Process, Cleanroom, Agile (XP, Scrum)
Software and Process Metrics
To **systematically** compare and **improve** industrial products, we need to precisely **describe and assess** the **products** and the **process of creation**.

This common practice for many **material** good, e.g. cars

- fuel consumption,
- size of trunk,
- fixed costs per year,
- time needed to change headlight’s light bulb,
- clearance (accuracy of fit and gaps of, e.g., doors)
- ...

**Note:** all these key figures are **models** of products — they reduce everything but the aspect they are interested in.

Less common practice for **immaterial** goods like Software.

It should be — (objective) **measures** are central to engineering approaches.

Yet: it's not that easy for software.
Excursion: Scales
Scales and Types of Scales

• measuring maps elements from a set $A$ to a scale $M$:

$$m : A \rightarrow M$$

• we distinguish

  (i) **nominal** scale

  • operations: $= \text{ (and } \neq \text{)}$

  (ii) **ordinal** scale

  • operations: $=, <, />$ (with transitivity), min/max, **percentiles** (e.g. median)

  (iii) **interval** scale (with units)

  • operations: $=, <, >, \text{ min/max, percentiles, } \Delta$

  (iv) **rational** scale (with units)

  • operations: $=, <, >, \text{ min/max, percentiles, } \Delta, \text{ proportion, } 0$

  (v) **absolute** scale

  • a rational scale where $M$ comprises the key figures itself
Nominal Scale

\[ m : A \rightarrow M \]

- operations: \( = \) (and \( \neq \))
- that is, there is no (natural) order between elements of \( M \),
- the lexicographic order can be imposed, but is not related to measured information (thus not natural).

**general example:**

- nationality, gender, car manufacturer, geographic direction, . . .
- Autobahn number, train number, . . .

**software engineering example:**

- programming language
**Ordinal Scale**

\[ m : A \rightarrow M \]

- operations: \( =, <, >, \min/\max, \text{percentiles} \) (e.g. median)
- there is a (natural) **order** between elements of \( M \), but no (natural) notion of **distance** or **average**

**general example:**
- strongly agree \( > \) agree \( > \) disagree \( > \) strongly disagree
- administrative ranks: Chancellor \( > \) Minister
- ranking list, leaderboard:
  finishing number tells us who was, e.g. faster, than who; but nothing about how much faster 1st was than 2nd
- types of scales, . . .

**software engineering example:**
- CMMI scale (maturity levels 1 to 5)
Interval Scale

\[ m : A \rightarrow M \]

- operations: \( =, <, >, \min/\max, \text{percentiles}, \Delta \)
- there’s a (natural) notion of difference \( \Delta : M \times M \rightarrow \mathbb{R} \),
- but no (natural) 0

- general example:
  - temperature in Celsius (no zero),
  - year dates,
    - two persons, born \( B_1, B_2 \), died \( D_1, D_2 \) (all dates beyond, say, 1900) — if \( \Delta(B_1, D_1) = \Delta(B_2, D_2) \), they reached the same age
- software engineering example:
  - time of check-in in revision control system,
Rational Scale

\[ m : A \to M \]

- operations: \( =, <, >, \text{min}/\text{max}, \text{percentiles}, \Delta, \text{proportion}, 0 \)
- the (natural) zero induces a meaning for proportion \( m_1/m_2 \)

- **general example:**
  - age ("twice as old"), finishing time, weight, pressure, …
  - price, speed, distance from Freiburg, …

- **software engineering example:**
  - runtime of a program for certain inputs,
Absolute Scale

\[ m : A \rightarrow M \]

- \( M = \mathbb{N}_0 \),
- a rational scale where \( M \) comprises the key figures itself
- absolute scale has median, but in general not an average in the scale.

**general example:**

- seats in a bus, number of public holidays, number of inhabitants of a country, \ldots
- “average number of children per family: 1.203” – what is a 0.203-child? the absolute scale has been viewed as a rational scale, makes sense for certain purposes

**software engineering example:**

- number of known errors,
Communicating Figures
**Median and Box-Plots**

<table>
<thead>
<tr>
<th></th>
<th>$M_1$</th>
<th>$M_2$</th>
<th>$M_3$</th>
<th>$M_4$</th>
<th>$M_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC</td>
<td>127</td>
<td>213</td>
<td>152</td>
<td>139</td>
<td>13297</td>
</tr>
</tbody>
</table>

- **arithmetic average**: 2785.6
- **median**: 127, 139, **152**, 213, 13297

- a **boxplot** visualises 5 aspects of data at once
  (whiskers sometimes defined differently, with “outliers”):

![Boxplot diagram](image)

100% (maximum)
75% (3rd quartile)
50% (median)
25% (1st quartile)
0% (minimum)

average: 7,033.027
median: 2,078

LOC lecture’s *.tex files
Software Metrics
**metric** — A quantitative measure of the degree to which a system, component, or process possesses a given attribute. See: quality metric.

IEEE 610.12 (1990)

**quality metric** — (1) A quantitative measure of the degree to which an item possesses a given quality attribute. (2) A function whose inputs are software data and whose output is a single numerical value that can be interpreted as the degree to which the software possesses a given quality attribute.

IEEE 610.12 (1990)
Definition. [Metric Space] Let $X$ be a set. A function $d : X \times X \rightarrow \mathbb{R}$ is called a metric on $X$ if and only if, for each $x, y, z \in X$,

(i) $d(x, y) \geq 0$ \hspace{1cm} \text{(non-negative)}
(ii) $d(x, y) = 0 \iff x = y$ \hspace{1cm} \text{(identity of indiscernibles)}
(iii) $d(x, y) = d(y, x)$ \hspace{1cm} \text{(symmetry)}
(iv) $d(x, z) \leq d(x, y) + d(y, z)$ \hspace{1cm} \text{(triangle inequality)}

$(X, d)$ is called a metric space.
Important **motivations** and **goals** for using software metrics:

- **Support** decisions
- **Quantify** experience, progress, etc.
- **Assess** the quality of products and processes
- **Predict** cost/effort, etc.

Metrics can be used:

- **descriptive** or **prescriptive**:
  - “the current average LOC per module is \( N \)” vs. “a procedure must not have more than \( N \) parameters”

- a **descriptive** metric can be **diagnostic** or **prognostic**:
  - “the current average LOC per module is \( N \)” vs. “the expected test effort is \( N \) hours”
  - **Note**: **prescriptive** and **prognostic** are different things.

- **Examples** for **diagnostic/guiding** use:
  - measure time spent per procedure before starting “optimisations”,
  - focus testing effort accordingly, e.g. guided cyclomatic complexity,
  - develop measures indicating architecture problems, (analyse,) then focus re-factoring
Requirements on Useful Metrics

**Definition.** A thing which is subject to the application of a metric is called **proband**. The value \( m(P) \) yielded by a given metric \( m \) on a proband \( P \) is called **valuation yield** (‘Bewertung’) of \( P \).

In order to be useful, a (software) metric should be:

- **differentiated** – worst case: same valuation for all probands
- **comparable** – ordinal scale, better: rational (or absolute) scale
- **reproducible** – multiple applications of a metric to the same proband should yield the same valuation
- **available** – valuation yields need to be in place when needed
- **relevant** – wrt. overall needs
- **economical** – worst case: doing the project gives a perfect estimation of duration, but is expensive; **irrelevant** metrics are not economical (if not available for free)
- **plausible** – (\( \rightarrow \) pseudo-metric)
- **robust** – developers cannot arbitrarily manipulate the yield; antonym: **subvertible**
## Requirements on Useful Metrics: Examples

<table>
<thead>
<tr>
<th>characteristic (‘Merkmal’)</th>
<th>positive example</th>
<th>negative example</th>
</tr>
</thead>
<tbody>
<tr>
<td>differentiated</td>
<td>program length in LOC</td>
<td>CMM/CMMI level below 2</td>
</tr>
<tr>
<td>comparable</td>
<td>cyclomatic complexity</td>
<td>review (text)</td>
</tr>
<tr>
<td>reproducible</td>
<td>memory consumption</td>
<td>grade assigned by inspector</td>
</tr>
<tr>
<td>available</td>
<td>number of developers</td>
<td>number of errors in the code (not only known ones)</td>
</tr>
<tr>
<td>relevant</td>
<td>expected development cost; number of errors</td>
<td>number of subclasses (NOC)</td>
</tr>
<tr>
<td>economical</td>
<td>number of discovered errors in code</td>
<td>highly detailed timekeeping</td>
</tr>
<tr>
<td>plausible</td>
<td>cost estimation following COCOMO (to a certain amount)</td>
<td>cyclomatic complexity of a program with pointer operations</td>
</tr>
<tr>
<td>robust</td>
<td>grading by experts</td>
<td>almost all pseudo-metrics</td>
</tr>
</tbody>
</table>

(Ludewig and Lichter, 2013)
Application domains for software metrics:

- **Cost** metrics (including duration)
- **Error** metrics
- **Volume/Size** metrics
- **Quality** metrics

Being **good** wrt. to a certain metric is in general not an asset on its own. In particular critical: pseudo-metrics for quality (→ in a minute).
**base measure** — measure defined in terms of an attribute and the method for quantifying it.

ISO/IEC 15939 (2011)

**Examples:**
- lines of code, hours spent on testing, . . .

**derived measure** — measure that is defined as a function of two or more values of base measures.

ISO/IEC 15939 (2011)

**Examples:**
- average/median lines of code, productivity (lines per hour), . . .
## Kinds of Metrics: by Measurement Procedure

<table>
<thead>
<tr>
<th></th>
<th>objective metric</th>
<th>subjective metric</th>
<th>pseudo metric</th>
</tr>
</thead>
<tbody>
<tr>
<td>Procedure</td>
<td>measurement, counting, poss. normed</td>
<td>review by inspector, verbal or by given scale</td>
<td>computation (based on measurements or assessment)</td>
</tr>
<tr>
<td>Advantages</td>
<td>exact, reproducible, can be obtained automatically</td>
<td>not subvertable, plausible results, applicable to complex characteristics</td>
<td>yields relevant, directly usable statement on not directly visible characteristics</td>
</tr>
<tr>
<td>Disadvantages</td>
<td>not always relevant, often subvertable, no interpretation</td>
<td>assessment costly, quality of results depends on inspector</td>
<td>hard to comprehend, pseudo-objective</td>
</tr>
<tr>
<td>Example, general</td>
<td>body height, air pressure</td>
<td>health condition, weather condition (“bad weather”)</td>
<td>body mass index (BMI), weather forecast for the next day</td>
</tr>
<tr>
<td>Example in Software Engineering</td>
<td>size in LOC or NCSI; number of (known) bugs</td>
<td>usability; severeness of an error</td>
<td>productivity; cost estimation following COCOMO</td>
</tr>
<tr>
<td>Usually used for</td>
<td>collection of simple base measures</td>
<td>quality assessment; error weighting</td>
<td>predictions (cost estimation); overall assessments</td>
</tr>
</tbody>
</table>

(Ludewig and Lichter, 2013)
### Some Objective Metrics, Base Measures

<table>
<thead>
<tr>
<th>dimension</th>
<th>name</th>
<th>unit</th>
<th>measurement procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>size of group, department, etc.</td>
<td>headcount</td>
<td>–</td>
<td>number of filled positions (rounded on 0.1); part-time positions rounded on 0.01</td>
</tr>
<tr>
<td>program size</td>
<td>–</td>
<td>LOC\textsubscript{tot}</td>
<td>number of lines in total</td>
</tr>
<tr>
<td>net program size</td>
<td>–</td>
<td>LOC\textsubscript{ne}</td>
<td>number of non-empty lines</td>
</tr>
<tr>
<td>code size</td>
<td>–</td>
<td>LOC\textsubscript{pars}</td>
<td>number of lines with not only comments and non-printable</td>
</tr>
<tr>
<td>delivered program size</td>
<td>–</td>
<td>DLOC\textsubscript{tot}, DLOC\textsubscript{ne}, DLOC\textsubscript{pars}</td>
<td>like LOC, only code (as source or compiled) given to customer</td>
</tr>
<tr>
<td>number of units</td>
<td>unit-count</td>
<td>–</td>
<td>number of units, as defined for version control</td>
</tr>
</tbody>
</table>

(Ludewig and Lichter, 2013)

- Note: **who** measures **when**?
### Assessment of Subjective Metrics

<table>
<thead>
<tr>
<th>kind of assessment</th>
<th>example</th>
<th>problems</th>
<th>countermeasures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Statement</td>
<td>“The specification is available.”</td>
<td>Terms are ambiguous, conclusions are hardly possible.</td>
<td>Allow only certain statements, characterise them precisely.</td>
</tr>
<tr>
<td>Assessment</td>
<td>“The module is coded in a clever way.”</td>
<td>No basis for comparisons.</td>
<td>Only offer particular outcomes, put them on an (at least ordinal) scale.</td>
</tr>
<tr>
<td>Grading</td>
<td>“Readability is graded 4.0.”</td>
<td>Subjective, grading not reproducible.</td>
<td>Define criteria for grades; give examples how to grade</td>
</tr>
</tbody>
</table>

(Stanewig and Lichter, 2013)
Some Subjective Metrics

- **Norm Conformance**
  Considering (all or some of)
  - size of units (modules etc.)
  - labelling
  - naming of identifiers
  - design (layout)
  - separation of literals
  - style of comments

- **Locality**
  - use of parameters
  - information hiding
  - local flow of control
  - design of interfaces

- **Readability**
  - data types
  - structure of control flow
  - comments

- **Testability**
  - test driver
  - test data
  - preparation for test evaluation
  - diagnostic components
  - dynamic consistency checks

- **Typing**
  - type differentiation
  - type restriction

(Ludewig and Lichter, 2013)
Practical Use of Grading-based Metrics

- Grading by human inspectors can be used to construct sophisticated grading schemes, see (Ludewig and Lichter, 2013).

- Premises for their practical application:
  - **Goals and priorities** are fixed and **known** (communicated).
  - **Consequences** of the assessment are **clear and known**.
  - **Accepted inspectors** are fixed.
  - The inspectors **practiced** on existing examples.
  - **Results of the first try** are **not over-estimated**, procedure is improved before results becoming effective.
  - Also **experienced developers** work as **inspectors**.
  - **Criteria and weights** are **regularly checked** and adjusted if needed.
Pseudo-Metrics
Some of the **most interesting aspects** of software development projects are **hard or impossible** to measure directly, e.g.:

- is the **documentation** sufficient and well usable?
- how much **effort** is needed until completion?
- how is the **productivity** of my software people?
- how **maintainable** is the software?
- do all modules do **appropriate error handling**?

Due to **high relevance**, people **want to measure** despite the difficulty in measuring. Two main approaches:

<table>
<thead>
<tr>
<th>Expert review, grading</th>
<th>differentiated</th>
<th>comparable</th>
<th>reproducible</th>
<th>available</th>
<th>relevant</th>
<th>economical</th>
<th>plausible</th>
<th>robust</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✓!</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pseudo-metrics, derived measures</th>
<th>differentiated</th>
<th>comparable</th>
<th>reproducible</th>
<th>available</th>
<th>relevant</th>
<th>economical</th>
<th>plausible</th>
<th>robust</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓!</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>
**Pseudo-Metrics Cont’d**

**Note:** not every derived measure is a pseudo-metric:

- **average lines of code per module**: derived, not pseudo
  → we really measure average LOC per module.

- use average lines of code per module to measure **maintainability**: derived, pseudo
  → we don’t really measure maintainability;
    average-LOC is only interpreted as maintainability.
    Not robust, easily subvertible (see exercises).

**Example:** **productivity** (derived).

- Team $T$ develops software $S$ with LOC $N = 817$ in $t = 310$ h.
- Define **productivity** as $p = \frac{N}{t}$, here: ca. 2.64 LOC/h.
- Pseudo-metric: measure **performance, efficiency, quality**, ... of teams by productivity (as defined above).

```
x := y + z;
```

- team may write $x := y + z$; instead of $x := y + z$; → 5-time productivity increase, real efficiency actually decreased.
Still, pseudo-metrics can be useful if there is a correlation with few false positives and false negatives between valuation yields and the property to be measured:

\[
\begin{array}{c|cc}
\text{valuation yield} & \text{low} & \text{high} \\
\hline
\text{high} & \text{false positive} & \text{true positive} \\
\text{low} & \text{true negative} & \text{false negative} \\
\end{array}
\]

Which may strongly depend on context information:

- if everybody adheres to a certain coding style, LOC says “lines of code in this style” — this may be a useful measure.
**McCabe Complexity**

**complexity** — (1) The degree to which a system or component has a design or implementation that is difficult to understand and verify. Contrast with: simplicity.
(2) Pertaining to any of a set of structure-based metrics that measure the attribute in (1).

**IEEE 610.12 (1990)**

---

**Definition.** [Cyclomatic Number [graph theory]] Let $G = (V, E)$ be a graph comprising vertices $V$ and edges $E$. The **cyclomatic number** of $G$ is defined as

$$v(G) = |E| - |V| + 1.$$  

**Intuition:** minimum number of edges to be removed to make $G$ cycle free.
Definition. [Cyclomatic Complexity [McCabe, 1976]] Let \( G = (V, E) \) be the Control Flow Graph of program \( P \). Then the cyclomatic complexity of \( P \) is defined as \( v(P) = |E| - |V| + p \) where \( p \) is the number of entry or exit points.

```java
void insertionSort(int[] array) {
    for (int i = 2; i < array.length; i++) {
        int j = i;
        while (j > 0 && array[j] < array[j - 1]) {
            array[j] = array[j - 1];
            j--;
        }
        array[j] = tmp;
    }
}
```

Number of edges: \( |E| = 11 \)
Number of nodes: \( |V| = 6 + 2 + 2 = 10 \)
External connections: \( p = 2 \)

\[ v(P) = 11 - 10 + 2 = 3 \]
Definition. [Cyclomatic Complexity [McCabe, 1976]] Let $G = (V, E)$ be the Control Flow Graph of program $P$. Then the cyclomatic complexity of $P$ is defined as $v(P) = |E| - |V| + p$ where $p$ is the number of entry or exit points.

- **Intuition**: number of paths, number of decision points.
- **Interval scale** (not absolute, no zero due to $p > 0$); easy to compute
- Somewhat independent from programming language.
- **Plausibility**: doesn’t consider data.
- **Plausibility**: nesting is harder to understand than sequencing.
- **Prescriptive** use: “For each procedure, either limit cyclomatic complexity to [agreed-upon limit] or provide written explanation of why limit exceeded.”
### Code Metrics for OO Programs (Chidamber and Kemerer, 1994)

<table>
<thead>
<tr>
<th>Metric</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weighted methods per class (WMC)</td>
<td>[ \sum_{i=1}^{n} c_i, n = \text{number of methods}, c_i = \text{complexity of method } i ]</td>
</tr>
<tr>
<td>Depth of inheritance tree (DIT)</td>
<td>Graph distance in inheritance tree (multiple inheritance?)</td>
</tr>
<tr>
<td>Number of children of a class (NOC)</td>
<td>Number of direct subclasses of the class</td>
</tr>
<tr>
<td>Coupling between object classes (CBO)</td>
<td>[ CBO(C) =</td>
</tr>
<tr>
<td>Response for a class (RFC)</td>
<td>[ RFC =</td>
</tr>
<tr>
<td>Lack of cohesion in methods (LCOM)</td>
<td>[ \max(</td>
</tr>
</tbody>
</table>

- **Objective metrics:** DIT, NOC, CBO; **pseudo-metrics:** WMC, RFC, LCOM

... there seems to be agreement that it is far more important to focus on empirical validation (or refutation) of the proposed metrics than to propose new ones, ... *(Kan, 2003)*
Goal-Question-Metric
Goal-Question-Metric (Basili and Weiss, 1984)

The three steps of **GQM**:

(i) Define the **goals** relevant for a project or an organisation.

(ii) From each goal, derive **questions** which need to be answered to check whether the goal is reached.

(iii) For each question, **choose** (or develop) metrics which contribute to finding answers.

**Note**: we usually want to optimise wrt. **goals**, not wrt. **metrics**.

**Development of pseudo-metrics**:

(i) Identify **aspect** to be represented.

(ii) Devise a **model** the aspect.

(iii) Fix a **scale** for the metric.

(iv) Develop a **definition** of the pseudo-metric, how to compute the metric.

(v) Develop **base measures** for all parameters of the definition.

(vi) **Apply** and **improve** the metric.
Now, Which Metric Should We Use?

It is often useful to collect some basic measures before they are actually required, in particular if collection is cheap:

- **size**
  - of newly *created* and *changed code*,
  - of separate *documentation*,
- **effort**
  - for *coding, review, testing, verification, fixing, maintenance*, . . .
  - for *restructuring* (preventive maintenance), . . .
- **errors**
  - at least errors *found* during quality assurance, and errors *reported* by customer
  - for *recurring problems* causing *significant effort*:
    - is there a (pseudo-)metric which correlates with the problem?

**Measures derived** from the above basic measures:

- *error rate* per release, *error density* (errors per LOC),
- average effort for error *detection* and *correction*,
- . . .

If in doubt, use the simpler measure.
Now, Which Metric Should We Use?

It is often useful to collect some basic measures before they are actually required, in particular if collection is cheap:

- **size**
  - of newly created and changed code,
  - of separate documentation,
- **effort**
  - for coding, review, testing, verification, fixing, maintenance, ...
  - for restructuring (preventive maintenance), ...
- **errors**
  - at least errors found during quality assurance,
  - for recurring problems causing significant effort: is there a (pseudo-)metric which correlates with the problem?

Measures derived from the above basic measures:

- error rate per release, error detection and correction effort
- average effort for error detection and correction
- ...

If in doubt, use the simpler measure.

LOC and changed lines over time (obtained by statsvn(1)).
References
References


