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:

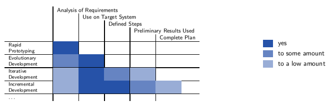
- process, model, process vs. procedure model
- code & fix, waterfall, S/P/E programs, (rapid) protoyping

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



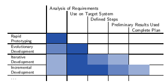
evolutionary software development — an approach which includes evolutions of the developed software under the influence of practical/field testing. New and changed requirements are considered by developing the software in **sequential steps of evolution**.

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

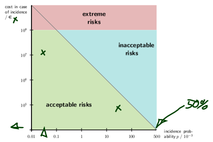
Quick Excursion: Risk and Riskvalue

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)

Barry W. Boehm

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

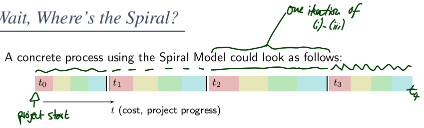
Advantages:

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

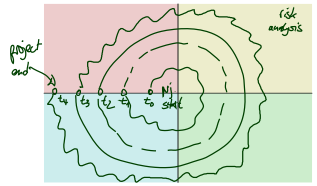
Note: **risk** can by anything; e.g. open technical questions (→ prototype?), but also lead developer leaving the company (→ invest in documentation), changed market situation (→ adapt appropriate features), …

Wait, Where's the Spiral?

A concrete process using the Spiral Model could look as follows:

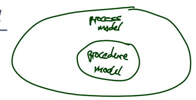


- fix goals, conditions, - risk analysis, - develop and test, - plan next phase,



Process Models

From Procedure to Process Model



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

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:

$$m : A \to M$$

- we distinguish
  - (i) **nominal** scale
    - operations: $=$ (and $\neq$)
  - (ii) **ordinal** scale
    - operations: $=$, $</>$ (with transitivity), $\min/\max$, **percentiles** (e.g. median)
  - (iii) **interval** scale (with units)
    - operations: $=$, $<$, $>$, $\min/\max$, percentiles, Δ
  - (iv) **rational** scale (with units)
    - operations: $=$, $<$, $>$, $\min/\max$, percentiles, Δ, proportion, 0
  - (v) **absolute** scale
    - a rational scale where M comprises the key figures itself

Nominal Scale

$$m : A \to M$$

- 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 laguage
  -

Ordinal Scale

$$m : A \to M$$

- operations: $=$, $<$, $>$, $\min/\max$, **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

$$m : A \to M$$

- operations: $=$, $<$, $>$, $\min/\max$, percentiles, Δ
- there's a (natural) notion of difference $\Delta : M \times M \to \mathbb{R}$,
- 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: $=, <, >, \min/\max$, percentiles, Δ, 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

$$m : A \to M$$

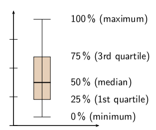
- $M = \mathbb{N}_0$,
- a rational scale where M comprises the key figures itself
- absolute scale has **median**, but in general not an average **in** the scale.
- **general example**:
  - seats in a bus, number of public holidays, number of inhabitants of a country, ...
  - "average number of children 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,
  -

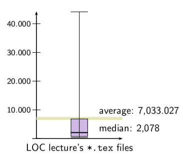
Communicating Figures

Median and Box-Plots

	M_1	M_2	M_3	M_4	M_5
LOC	127	213	152	139	13297

- **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"):





Software Metrics

Software Metrics

metric — A quantitative measure of the degree to which a system, component, or process posesses 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)**

Recall: Metric Space [math.]

Definition. [*Metric Space*] Let X be a set. A function $d : X \times X \to \mathbb{R}$ is called **metric** on X if and only if, for each $x, y, x \in X$,

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

(X, d) is called **metric space**.

Software Metrics: Motivation and Goals

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 prodecure must not have more then 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 estimatio of duration, but is expensive;
  irrelevant metrics are not economical (if not available for free)
- **plausible** – ($\to$ pseudo-metric)
- **robust** – developers cannot arbitrarily manipulate the yield; antonym: **subvertible**

Requirements on Useful Metrics: Examples

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

(Ludewig and Lichter, 2013)

Software Metrics: Blessing and Curse

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 ($\to$ in a minute).

Kinds of Metrics

Kinds of Metrics: ISO/IEC 15939:2011

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

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

(Ludewig and Lichter, 2013)

Some Objective Metrics, Base Measures

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

(Ludewig and Lichter, 2013)

- Note: **who** measures **when**?

Assessment of Subjective Metrics

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

(Ludewig 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

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:

	differentiated	comparable	reproducible	available	relevant	economical	plausible	robust
Expert review, grading	(✔)	(✔)	(✘)	(✔)	✔!	(✘)	✔	✔
Pseudo-metrics, derived measures	✔	✔	✔	✔	✔!	✔	✘	✘

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).

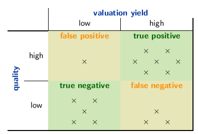
Example: **productivity** (derived).

- Team T develops software S with LOC $N = 817$ in $t = 310$h.
- Define **productivity** as $p = N/t$, here: ca. 2.64 LOC/h.
- Pseudo-metric: measure **performance**, **efficiency**, **quality**, ... of teams by productivity (as defined above).
- team may write
```
x
:=
y
+
z;
```
  instead of `x := y + z;`

→ 5-time productivity increase, real efficiency actually decreased.

Pseudo-Metrics Cont'd

- 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:



- 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

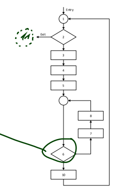
$$v(G) = |E| - |V| + 1.$$

Intuition: minimum number of edges to be removed to make G cycle free.

McCabe Complexity Cont'd

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**.

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



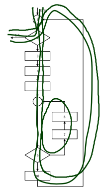
Number of edges: $|E| = 11$
Number of nodes: $|V| = 6 + 2 + 2 = 10$
External connections: $p = 2$

→ $v(P) = 11 - 10 + 2 = 3$

McCabe Complexity Cont'd

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."



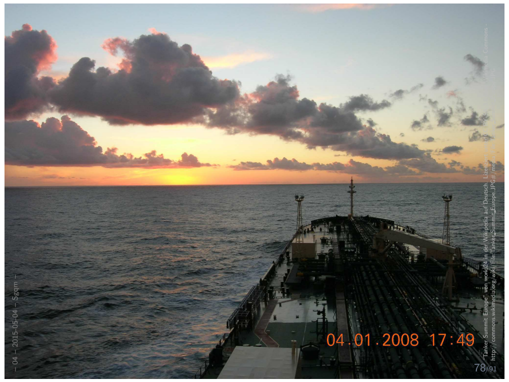
Code Metrics for OO Programs (Chidamber and Kemerer, 1994)

metric	computation
weighted methods per class (WMC)	$\sum_{i=1}^{n} c_i$, n = number of methods, c_i = complexity of method i
depth of inheritance tree (DIT)	graph distance in inheritance tree (multiple inheritance ?)
number of children of a class (NOC)	number of direct subclasses of the class
coupling between object classes (CBO)	$CBO(C) = \|K_o \cup K_i\|$, K_o = set of classes used by C, K_i = set of classes using C
response for a class (RFC)	$RFC = \|M \cup \bigcup_i R_i\|$, M set of methods of C, R_i set of all methods calling method i
lack of cohesion in methods (LCOM)	$\max(\|P\| - \|Q\|, 0)$, P = methods using no common attribute, Q = methods using at least one common attribute

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

> *... there seems to be angreement 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?

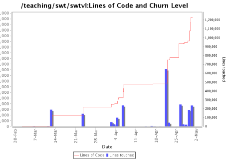
It is often useful to collect some basic measures before they are actually required, in particular if collection is cheap:

- **size**
  - of newly **created** and **changed code**,
  - of separate **documentation**,
- **effort**
  - for **coding**, **review**, **testing**, **verification**, **fixing**, **maintenance**, ...
  - for **restructuring** (preventive r
- **errors**
  - at least errors **found** during q
- for **recurring problems** causin
  is there a (pseudo-)metric wh

Measures derived from the a

- **error rate** per release, **error c**
- average effort for error **detect**
- ...

If in doubt, use the simpler m



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

References

References

Abrahamsson, P., Salo, O., Ronkainen, J., and Warsta, J. (2002). Agile software development methods. review and analysis. Technical Report 478.

Basili, V. R. and Weiss, D. M. (1984). A methodology for collecting valid software engineering data. *IEEE Transactions of Software Engineering*, 10(6):728–738.

Beck, K. (1999). *Extreme Programming Explained – Embrace Change*. Addison-Wesley.

Boehm, B. W. (1988). A spiral model of software development and enhancement. *IEEE Computer*, 21(5):61–72.

Chidamber, S. R. and Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493.

Hörmann, K., Dittmann, L., Hindel, B., and Müller, M. (2006). *SPICE in der Praxis: Interpretationshilfe für Anwender und Assessoren*. dpunkt.verlag.

IEEE (1990). *IEEE Standard Glossary of Software Engineering Terminology*. Std 610.12-1990.

ISO/IEC (2011). *Information technology Software engineering Software measurement process*. 15939:2011.

Kan, S. H. (2003). *Metrics and models in Software Quality Engineering*. Addison-Wesley, 2nd edition.

Ludewig, J. and Lichter, H. (2013). *Software Engineering*. dpunkt.verlag, 3. edition.

Schwaber, K. (1995). SCRUM development process. In Sutherland, J. et al., editors, *Business Object Design and Implementation, OOPSLA'95 Workshop Proceedings*. Springer-Verlag.

Team, C. P. (2010). Cmmi for development, version 1.3. Technical Report ESC-TR-2010-033, CMU/SEI.

V-Modell XT (2006). *V-Modell XT*. Version 1.4.

Züllighoven, H. (2005). *Object-Oriented Construction Handbook - Developing Application-Oriented Software with the Tools and Materials Approach*. dpunkt.verlag/Morgan Kaufmann.