Software Metrics, Software Measurement & Software Influences

'Concluding' Considerations

Reiner R. Dumke, University of Magdeburg, Germany
https://softmeasure.de

Abstract

AI, at the latest, demonstrates the almost totalitarian dependence on software systems in all areas of society. This article aims to show once again how software measurement has helped and can help to ensure successful and virtually conflict-free software use. Based on the (global) situation of the use of a wide variety of software systems in all areas of society, the main aim is to highlight the problems of software use.

In doing so, particular importance is attached to considering complexity in all its causes and facets. Based on our own experience with software systems, technologies, and paradigms over decades, a number of conclusions and experiences are postulated in conclusions.

1. The Cumulative Set of Current IT Technologies

1.1 Development Forms of Software Technologies and Paradigms

In the Programming Paradigms course, we defined this term as follows: "A programming paradigm encompasses the modeling, conceptual, and implementation fundamentals of a programming language" [Dumke 2003]. But readers can, of course, use their own terminology.

Based on our definition of a paradigm, which naturally also applies to software in general, software technology is then the implementation of a paradigm supported by special techniques, but above all by tools. Various methods and models have been used to implement software technology. Starting with an initial systematization known as the waterfall model, models such as the V-model, the spiral model, evolutionary development, and cleanroom engineering, through to agile ad hoc development, have been used, primarily to evaluate and quality-assure the results of a development phase [Sommerville 2010].

Of course, these software technologies are also tailored to the respective application, as we know them for real-time systems, web applications, or expert systems. However, these details are not so relevant here.

1.2 Examples of Software Technologies and Paradigms

I have worked in all this following described software technologies and paradigms (not only in practice but mostly in teaching) without any further explanations. The following technology overview is based on the author's experience with the development and application of software systems.

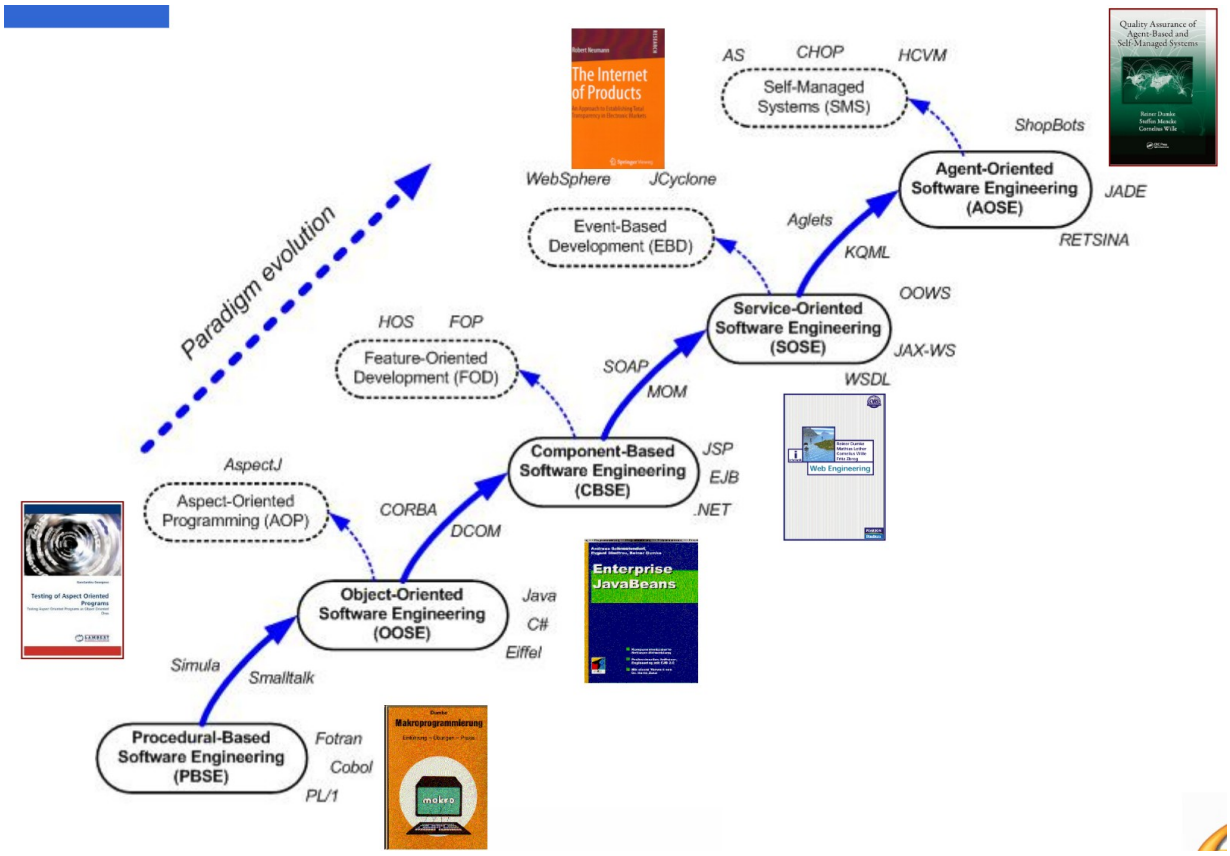

Fig. 1:The historical development of software paradigms

Especially, the paper of Harry Sneed in this Measurement News shows the existence and use of classical software paradigms currently.

Of course, decades of experience have also yielded parallels in the development of software development methods and technologies, as the following table shows [Dumke 2003].

Classical Methods	… leads to ...	Today's approach
Bosak' Information Algebra (60s) (also took into account queries about undefined or unsaved data)	*Extended by complex data, knowledge and image structures* ========>>	*Data Mining and Data Quality*
Attributed grammars and macro technique (70s)	*Extended by language-oriented attribute structures* ========>>	*Aspect-oriented Programming (AOP)*
Configuration management (70s)	*Extended by domains* ========>>	*Product lines*
Software generation, PDL, LOTOS (80s)	*Extended by component strucures* ========>>	*Model-driven Architecture (MDA)*
Rapid Application Development (RAD) (80s)	*Extended by service structures* ========>>	*Agile Development Methods*
Higher Order Software (HOS) (90s)	*Extended by dynamic architectures* ========>>	*Feature-oriented Programming (FOP)*

Tab. 1: The use of classical methods and approaches today

Furthermore, the component-based software engineering (CBSE) approach has demonstrated that the systems developed are based on a large number of components from system and application software from different providers and industries.

1.3 Our First Conclusion considering the Software Technologies

The situation facing software measurement from the perspective of software technologies leads us to our first conclusion in the following form:

> ***In today's IT world, almost all existing software technologies coexist, and in some cases even within the same area of application. This alone illustrates the almost 'chaotic' situation surrounding the application of software measurement and evaluation methods.***

2. The Particular Influence of Software Complexity

2.1 The Different Kinds of Program Complexities

A typical complexity point of view in computer science is the algorithm related O notation [Leiss 2007] and any kinds of algorithmic complexity [Hemaspaandra 2002].

But, Capers Jones see – during the measurement of different software systems – the different kinds of complexity considering many kinds of program paradigms and could be used for system evaluation to all variants of software infrastructures [Jones 2007]:

Algorithmic complexity: "concerns the length and structure of the algorithms for computable problems." Typical classifications based on the *Rogues' gallery of complexity classes* are P, NP, PSPACE, EXP, FP, SSF etc. [Hemaspaandra 2002].

Code complexity: "concerns the subjective views of development and maintenance personnel about whether the code they are responsible for is complex or not." This complexity could be measured by counting of structures and flows in the source code [Abran 2006].

Combinatorial complexity: "concerns the numbers of subsets and sets that can be constructed out of N components." Especially, the distributed systems meet this kind of complexity in *test cases and test automation* [Farooq 2009].

Computational complexity: "concerns the amount of machine time and the number of iterations required to execute an algorithm." A typical characterization of this complexity is based on the so-called O notation [Leiss 2007].

Cyclomatic complexity: "is derived from graph theory and was popular for software by Tom McCabe." This complexity can be used for estimation of test effort and/or maintenance effort as well [Fenton 1997].

Data complexity: "deals with the number of attributes associated with entities." Its importance has been increased using complex network technologies like *Grid or Cloud Computing* [Bhowmick 2004].

Diagnostic complexity: "is derived from medical practice, where it deals with the combinations of symptoms." For software systems, this complexity was considered using a set of measures presented in kiviat diagrams [Ebert 2004].

Entropic complexity: "is the state of disorder of the component parts of a system." Especially in large scale system, this complexity leads to efforts in maintaining and evolution [Singpurwalla 1999].

Essential complexity: "is also derived from graph theory and was mode popular by Tom McCabe." The intention is to estimate the structure conformity based on a principle like structured programming [Zuse 1998].

Fan complexity: "refers to the number of times of software module is called (termed fin in) or the number of modules that it calls (termed fan out)." This complexity deals with *component structures* [Lorenz 1994].

Flow complexity: "is a major topic in the studies of fluid dynamics and meteorology. It deals with the turbulence of fluids moving through channels and across obstacles." This complexity could be used for characterization the *Web service application and orchestration* [Dumke 2008].

Function point complexity: "refers to the set of adjustment factors needed to calculate the final adjusted function point total of a software project." This aspect is an essential part of effort/cost estimation based on function point counting [Dumke 2011].

Graph complexity: "is derived from graph theory and deals with the numbers of edges and nodes on graphs created for various purposes." Against the cyclomatic and essential complexity, this kind o complexity involves more graph-based measurements [Zuse 1998].

Halstead complexity: "is derived from the *Software Science* research carried out by the late Maurice Halstead." This complexity deals with the number of syntactical elements like operators and operands and is based on the assumption that a lot of operators/operands and a large variance of operators/operands implies a larger effort of implementation [Kitchenham 1996].

Information complexity: "is concerned with the numbers of entities and the relationships between them that might be found in a database, data repository, or data warehouse." This complexity is one of the main characteristic in *Cloud Computing efficiency* [Whitmire 1997].

Logical complexity: "is important for both software and circuit design. It is based up on the combinations of AND, OR, NOR, and NAND logic conditions that are concatenated together." *Service choreography* could involve such kind of complexity [Dumke 2008].

Mnemonic complexity: "is derived from cognitive psychology and deals with the ease or difficulty of memorization." This complexity could be appeared in system debugging or system analysis [Munson 2003].

Organizational complexity: "deals with the way human beings in corporations arrange themselves into hierarchical groups or matrix organizations." The new kind of distributed system development and evolution implies this kind of complexity for the management processes [Jones 2010].

Perceptional complexity: "is derived from cognitive psychology and deals with the arrangements of edges and surfaces that appear to be simple or complex to human observers." This complexity is one of the essential aspect for *usability of software systems* [Munson 2003].

Problem complexity: "concerns the subjective views of people asked to solve various kinds of problems about their difficulty." A special kind of this

complexity is described above as Lehman classification [Pandian 2004].

Process complexity: "is mathematically related to flow complexity, but in day-to-day software work it is concerned with the flow of materials through a software development cycle." Considering the different kinds of software development indifferent paradigms, this complexity has been extended until today dramatically [Putnam 2003].

Semantic complexity: "is derived from the study of linguistics and is concerned with ambiguities in the definitions of terms." Especially in Web service orchestration, we could identify this kind of complexity [Mendes 2006].

Syntactic complexity: "is also derived from linguistics and deals with the grammatical structure and lengths of sections, such as sentences and paragraphs." Plenty of syntactical related metrics for programming languages exists and are used for quality assurance [Zuse 1998].

Topologic complexity: "deals with rotation and folding patterns." Especially in *(social) networks,* this complexity must be considered for measurement and evaluation of performance and efficiency [Dumke 2001].

This multitude of types and forms of complexity illustrates above all the complex situation and immense challenge of software system development.

2.2 The Growth of Software Complexity

Not only is computer performance increasing at an almost exponential rate, but the complexity of software technologies and thus also of software systems is also increasing significantly.

Since the inception of the Java language, the number of predefined classes has risen from an initial 212 classes (Java 1.0) to 1,842 classes (Java 1.3) and then to over 5,000 classes (Java 1.8) at the beginning of the 2000s. When we introduced Java training at our university in 1996, we also recognized the challenges posed by this increasing complexity in programming language training [Dumke 2003].

After all, who knows the nearly 6,000 classes with over 60,000 methods and can apply them sensibly? This then led to simplified learning methods for these comprehensive programming languages such as 'learning by doing' or 'learning by example'.

Not only within a technology, but also in terms of further development, there has been a significant increase in complexity.

From OOSE to CBSE: A comparison of the coupling factors between Java objects or classes and CORBA revealed a fivefold increase [Dumke 2002].

From CBSE to SOSE: Measurements of selected web services revealed twice as much *interoperability* between components as in component-based system solutions [Schmietendorf 2007].

From SOSE to AOSE: The measurement and evaluation of the agent system 'Agent Academy' as a European project revealed consistent quality characteristics with virtually unchanged complexity despite increasing complexity [Wille 2005].

2.3 The Inevitability of Software Errors

The existence of complexity in all dimensions of a software-based system ultimately formed the basis of our so-called iceberg model, which illustrates "visibility" in the world of computer science [Dumke 2014].

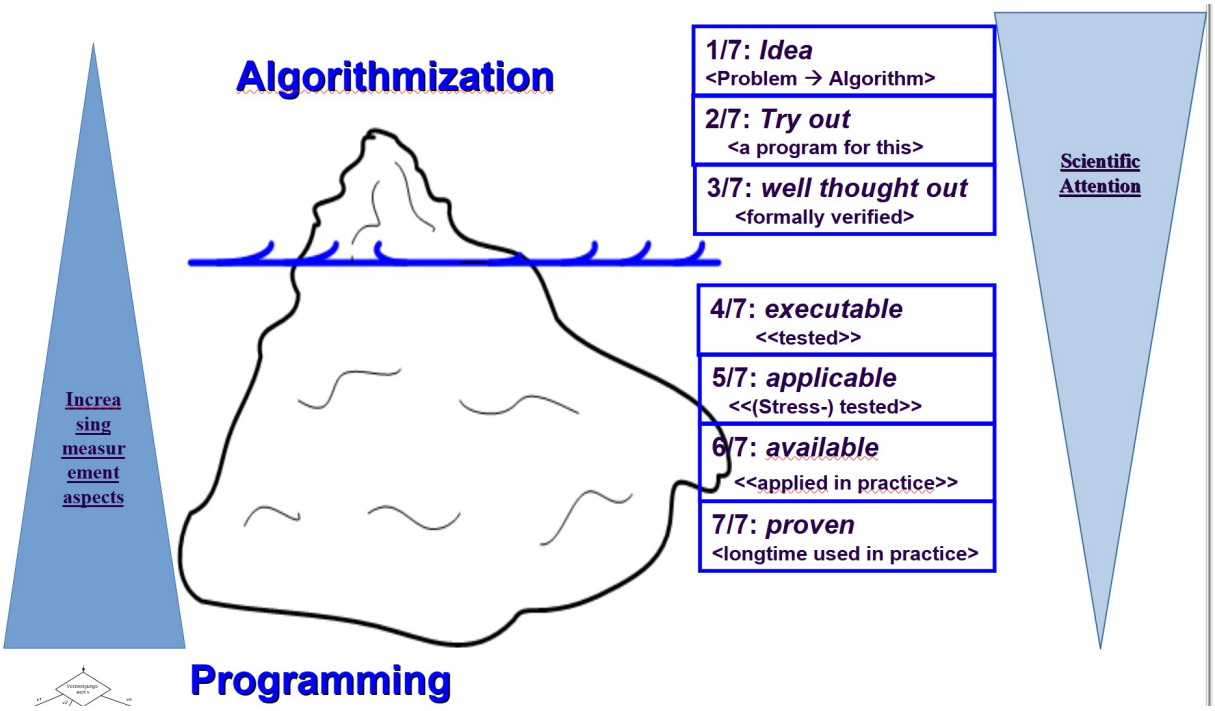

Fig. 2: Our iceberg model of algorithmization and programming

Typically, 3/7 of the iceberg is visible (an "ideal world" of algorithm theory) and 4/7 is the "dark side" of programming, which is almost impossible to control (*in the sense of being completely free of errors*).

The impossibility of achieving flawlessness is not due to negligence on the part of the programmers during implementation, but rather to the almost unmanageable complexity of a comprehensive software implementation. Respecting and taking this complexity into account is therefore an important principle in software development and its application [Basili 1984]. This is also the reason why application experience plays an important role for software systems.

This naturally leads to the ***general software problem*** in the form:

At what level of software complexity does formally verifiable representation transition into a probability-based system analysis?

This is also the basis for the error estimates for software, which we have presented for comprehensive software systems in the following table ([Genuchten 2013], [Dumke 2023]).

Software System	***Lines of Code***	***Average estimated Errors*** ***(by Halstead, Jones & Sneed)***
Average iPhone App	40,000	210
Space Shuttle Software	400,000	1,801
Hubble Telescope	2,000,000	9,009
Photoshop C.S. 6	4,000,000	18,019
DVD Player on xBox	4,500,000	20,272
Google Crome	5,300,000	23,875
Boeing 787	13,000,000	58,563
F-35 Fighter Jet	22,000,000	99,107
Microsoft Office 2013	44,000,000	198,215
Facebook	62,000,000	279,303
Car Software	100,000,000	450,490
Google	2,000,000,000	9,009,803

Tab. 2: Error estimates for large software systems

However, the advantage of a program or software is that it does not wear out and only undergoes changes in its properties through active maintenance. Hence, this also means that correct program sections remain correct if they are not changed.

On the other hand, software systems are always a conglomerate of various integrated components at different architectural levels, such as

- Components from the respective operating system or the platform used in each case ([Brown 1998], [Khalidi 2011] et al.),
- Components from other integrated systems based on the web technology used in each case ([Dumke 2003], [Menasce 1998] et al.),
- Components from ad hoc selected services within the framework of a flexible range of applications ([Cooper 2010], [Schmietendorf 2002] et al.),

This means the integration of software whose quality is based solely on empirical values.

2.4 Our Second Conclusion considering the Software Errors

The above explanations show that it is impossible to verify that software is completely error-free. This leads us to the following conclusion:

The implementation of software systems must cope with and master a wide variety of complexities, which means that it is impossible to guarantee that the application will be error-free.

3. The Influence of Empirical Aspects in Software Systems

3.1 The Reason for an Empirical Evaluation of the Software

The potential influence of empirical factors in software development and application was first discussed and motivated by Brooks (Mythical Man Month [Brooks 1975]) and Shneiderman (Software Psychology [Shneiderman 1980]).

Let us start with a simple consideration. In computer science, program creation is described as follows:

*Given a program **P** in the programming language **L** for an algorithm or specification **S**, in short*

$$S \dashrightarrow P(L)$$

However, the reality of program or software development can also be comprehensively described in the following form:

*A program **P** developed by developer **W** with training **A** and experience **E** from department **I** of company **F** in programming language **L** with program library **B** in programming time **T** and development method **M** according to the current (modern) paradigm **G**, the test procedure **V**, with the hardware **H** from manufacturer **R** based on an underlying algorithm **S** with complexity **K** and quality **Q** for customer **U,** or not quite so short*

$$S_K \dashrightarrow^{W(F,I,A,E),\ M(G,V),\ H(R),\ T} \dashrightarrow P_{L(B),Q(U)}$$

Extensive experiments, investigations, and studies have now been conducted to examine the possible effects of all these influencing factors—as ***empirical indicators***, so to speak.

On this basis, extensive empirical studies have been conducted and published to date ([Jones 2010], [Kitchenham 2007], [Sneed 2010], and many others). Furthermore, institutes such as the Software Engineering Institute in Pittsburgh

[Humphrey 1990] and the Software Engineering Laboratory in Maryland [Basili 2002] were founded.

In addition, an international collection of empirical software data was finally compiled in the so-called ISBSG (International Software Benchmark Standard Group) [ISBSG 2003]. Readers are certainly familiar with other forms of collecting and disseminating empirical data.

3.2 Examples for Empirical Evaluations of the Software

The following table provides useful insights into selected empirical data for the comprehensive software systems mentioned above.

Software System	*Lines of Code*	*Estimated Time of Program Verification [Personal Month]*	*Test Effort [Personal Month]*
Average iPhone App	40,000	111	40
Space Shuttle Software	400,000	45,352	7,040
Hubble Telescope	2,000,000	226,762	35,200
Photoshop C.S. 6	4,000,000	453,524	70,400
DVD Player on xBox	4,500,000	510,215	79,200
Google Crome	5,300,000	600,920	93,280
Boeing 787	13,000,000	1,473,954	228,800
F-35 Fighter Jet	22,000,000	2,494,383	387,200
Microsoft Office 2013	44,000,000	4,988,766	774,400
Facebook	62,000,000	7,029,625	1,091,200
Car Software	100,000,000	11,338,105	1,760,000
Google	2,000,000,000	226,762,105	35,200,000

Tab. 3: Empirical data on the effort required for program verification and the general testing effort for the above-mentioned large-scale software systems

Please don't argue now: I'm taking a team of 200,000 IT developers and… Furthermore, the impossibility of realizing such time expenditures naturally leads to abbreviated techniques and methods of software development that, in principle, ensure quality.

In addition, different values for the above-mentioned variables W, A, E, I, M, H, etc. are given for each of the specified lines of code, so that the comparison can be made on an ordinal rather than a ratio scale. Nevertheless, the evaluations illustrate the scope of complex software development. Empirical studies on software development, maintenance, and application yielded helpful results, such as

- The empirical analysis of a *goto statement* in programming languages revealed its problem [Dijkstra 1968] and ultimately led to the development of “structured programming languages” such as Modula and Algol 68. Because

of the dominance of the C world today, these principles no longer apply in Java, for example [Dumke 2003].

- An empirical analysis of the so-called *Personal Software Process* ([Humphrey 2000], see also [Richter 2015]) ultimately led to a development methodology for quality-assured software development, which, however, was reduced to an ad hoc methodology by time-optimized agile development.
- In particular, empirical studies on the *performance of service, agent*, *and other technologies l*ed to respective efficiency improvements in these areas ([Dumke 2010], [Dumke 2001], [Rud 2009] et al.).
- Extensive project analyses ultimately led to *reusable evaluations and tools* for almost the entire IT sector ([Boehm 2007], [ISBSG 2003], [Jones 2010], [McConnel 2006], et al.).

Above all, however, empirical studies have yielded important findings for successful *project management* with the necessary control and security techniques ([Binder 2000], [Bundschuh 2008], [Davis 2005], [Keyes 2003], [Sommerville 2010] et al.).

The following figures from a project dashboard in my *Software Estimation Apps* provide an overview of empirical evaluations of software development [Dumke 2023].

Fig.3: Examples of empirical evaluations of software development as dashboard and metrics overview using the *SoftwareExpert App* and evaluate this app itself

3.3 Our Third Conclusion motivating the Empirical Software Engineering

The problems with software development and the resulting software properties already mentioned above lead us to supplement classic software engineering with the following conclusion:

> ***The methods, approaches, technologies, and tools for systematic software development have become established as Software Engineering. Empirical analyses and presentations of results are indispensable for making this process understandable, controllable, and, in the broadest sense, manageable. This leads to the necessity of Empirical Software Engineering.***

4. Characterization of Software Measurement and Evaluation

4.1 A shortly Characterization of Software Measurement

Of course, it is not possible to provide a satisfactory overview of software measurement here. Nevertheless, we would like to outline our area of research as an example (a short overview you can find in our Measurement News in [Dumke 2021]).

After the beginnings of primarily program code metrics in the 1970s and 1980s, the introduction of measurement theory for software metrics and measures by Fenton [Fenton 1993] finally led to the most comprehensive and mathematically sound representations of software metrics by Zuse [Zuse 1998].

Further overviews of metrics supplemented the collection of measurement approaches for all areas of software development and maintenance ([Fenton 1997], [Kitchenham 1996], [Munson 2003] [Sneed 2005] et al.).

Specific instructions for the successful introduction of software measurement approaches in the context of software development can be found, for example, in [Bundschuh 2008] and [McGarry 2002] as implementation of the ISO 15939 standard, or for consistent alignment with business objectives in [Boehm 2007], [Pandian 2004], and [Putnam 2003].

Not to be forgotten are the experience-based approaches for the successful introduction of software measurement and evaluation in our community. One example is the so-called E4 approach, which represents the consistent and successful integration of software measurement into the business processes [Ebert 2007].

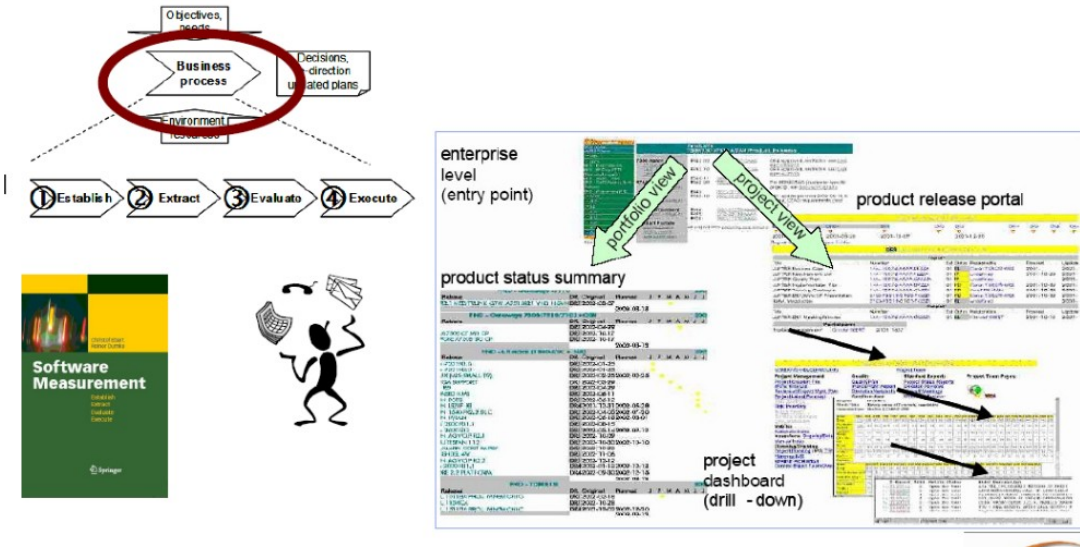

Fig. 4: Outline of the E4 software measurement approach

Another approach developed within our community describes the method for defining and practically introducing basic software metrics as standard measure in the general IT world [Abran 2010].

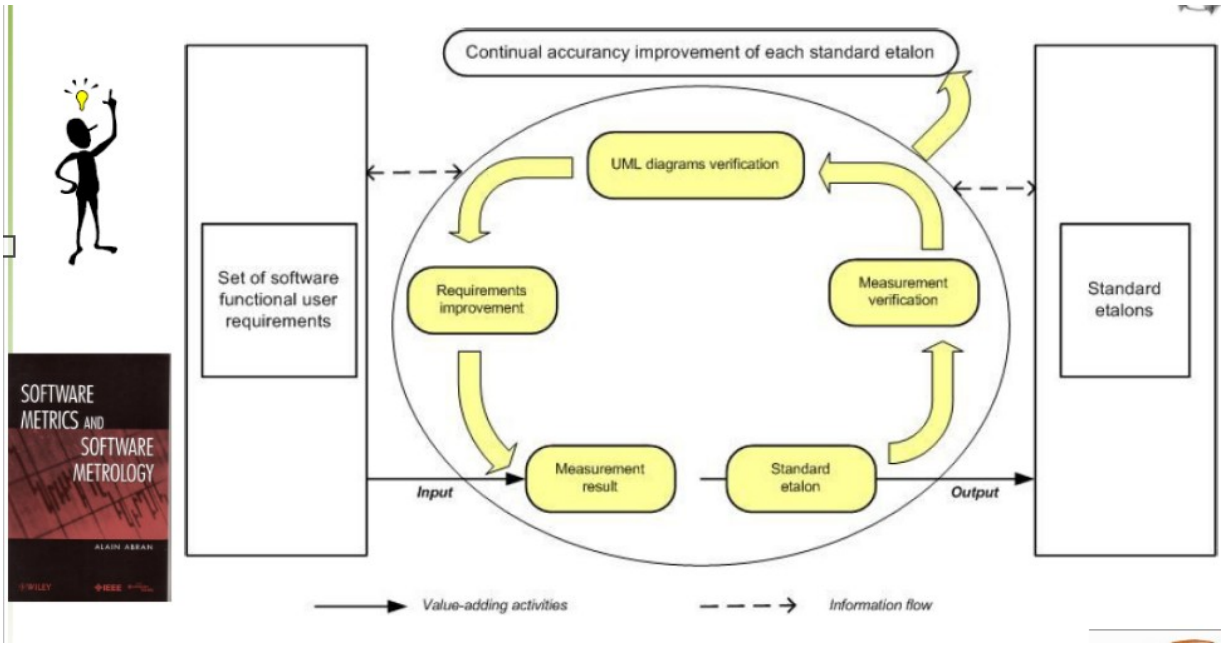

Fig. 5: Outline of the software measure definition and introduction

This approach forms the basis for the popular use of the COSMIC-FP method for determining the scope of software systems in a wide variety of application areas ([Abran 2006], [Dumke 2010]).

Finally, there is our own measurement approach, which has primarily been used for teaching purposes as the so-called CAME approach [Dumke 2008].

This measurement approach serves above all to characterize the level of software measurement of software products or processes in more detail and thus to identify areas of software development and application that have not been measured or controlled.

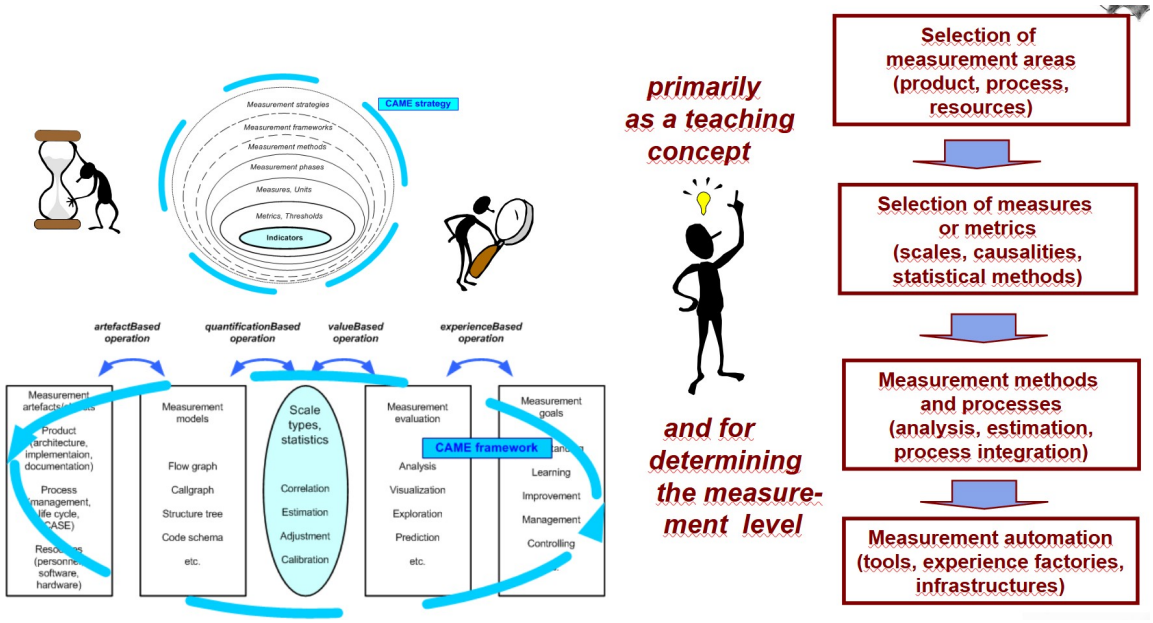

Fig. 6: Outline of our software measurement evaluation CAME approach

Of course, readers will be familiar with other important areas, aspects, and findings relating to software measurement, but this small selection will suffice for the purposes of this presentation.

4.2 Problems with Software Measurement and Evaluation

A very simplified representation of software development shows the cause of the problems mentioned here regarding software measurement and evaluation.

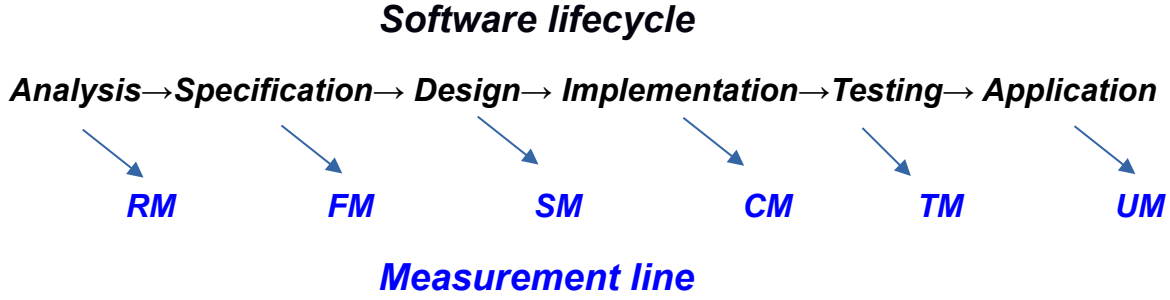

Fig. 7: Software development measurement point of views

Where means: RM – requirements measurement, FM – functionality measurements, SM – software mapping measurement, CM – code measurement, TM – test coverage measurement and UM – usability measurement.

The illustration is intended to clarify that software measurement and evaluation are always performed *retrospectively or additionally*. This means that, under time pressure, qualitative details may be analyzed retrospectively or omitted entirely.

In 2011 we identify the following problems in the application of software measurement and evaluation ([Büren 2011]):

1. ***The ignorance problem***: Software, and software processes in particular, are still largely unmeasured and therefore cannot be evaluated.

2. ***The motivation problem***: Software measurement is mostly voluntary and is still too often replaced by blanket assumptions and guidelines.

3. ***The quality problem***: Software measurement itself is still not sufficiently established as a universally applicable form of quality determination and assurance.

4. ***The criticality problem:*** The determination of the measurement objectives themselves is often still insufficiently defined in their specific and general context.

5. ***The complexity problem***: The artifacts to be measured (product, process, resources) are still insufficiently described in terms of their complexity for measurement.

6. ***The persistence problem:*** Software measurement methods are still not sufficiently integrated into development, maintenance, application methods, and tools.

7. ***The basic measurement unit problem***: There are hardly any metrics or even measurements for general comparability across products and processes.

8. ***The experience problem***: The thresholds for metrics or even units of measurement do not take sufficient account of paradigms or technology-related factors.

9. ***The causality problem***: Software measurement still takes too little account of both methodological and technological processes.

10. ***The evidence problem:*** Software engineering is still a long way from being axiomatic theory.

However, this list of problems was intended as a call to action and not as a fundamental problem with the obviousness of software measurement and evaluation in general.

This means that Basili's contribution to the Software Engineering Lab (SEL) and its termination shows that profitable research no longer seemed possible, rather than that software measurement is generally not useful [Basili 2002].

Similarly, DeMarco's general questioning of software measurement and software engineering is more of a provocative outline from a management perspective and a very superficial description in terms of software technology, and thus a somewhat oversimplified representation that was perhaps only intended to attract attention [DeMarco 2009].
.

4.3 Software Measurement in the ChatGPT

It was a wonderful experience to use ChatGPT to calculate code metrics, for example, and explain the results with AI, but it became apparent that

- Currently, ChatGPT is not familiar with any software measurement processes or technologies. Its presentation style can best be described as traditional.
- Furthermore, it does not distinguish between key figures or metrics and measurements with units of measurement.
- In addition, ChatGPT has no knowledge of research priorities or research institutions and individuals worldwide.
- There is hardly any information available about our community, and there is a complete lack of information about software scope measurement standards.

However, ChatGPT is, of course, also a learning system, so its limited knowledge of our field may improve in future.

4.4 Current Situation of Software Measurement and Evaluation

In addition to the problems that are certainly also encountered in other scientific disciplines, we would now like to present a positive 10-point list for the situation in the field of software measurement and evaluation:

1. Software Metrics help to ***understand the complexity of software (based) systems*** and applications and are used in different manners in practice.
2. Software measurement leads to a ***better understanding of software development*** characteristics like effort, costs and time.
3. Software measurement introduces a ***better definition of software components*** and processes.
4. Software experiments shows helpful ***relationships between measures*** like the size measures as FFP, Cosmic FP, LOC, UCP etc.
5. Software measurement creates a strong ***software measurement community*** in order to apply empirical software engineering in all kinds of applications.

6. Our software measurement community has produced ***essential measurement publications*** like books, journals, proceedings and web services.

7. Our software measurement community has introduced ***essential measurement process applications*** in order to manage software in general.

8. Our software measurement ***community was active world-wide*** e. g. in their conferences in four continents and many countries world-wide.

9. Our community has motivated in the future by ***qualification of students, practioneers and researchers.***

10. Software measurement has ***developed a strong research area of empirical software engineering.***

4.5 Our Forth Conclusion considering the Software Measurement

Our final conclusion summarizes the situation regarding software measurement and evaluation based on many years of experience:

> ***Software measurement and evaluation have had a significant impact on the development, maintenance, and application of software systems. From an initial understanding of systems, processes, and problems, integral control processes have ultimately been applied to software development and use. In addition, software measurement has led to empirical software engineering, which has clearly demonstrated a verifiable improvement in software quality but also the limits of absolute reliability.***

5 Conclusions

This article is an attempt to describe the situation in the field of software measurement and evaluation, or rather its general scientific manifestation as empirical software engineering.

Of course, it is hardly possible to cover this extensive field of knowledge in its entirety. It should only serve to show which conclusions (in brief) from a working phase of over 50 years in this field appear to be particularly relevant or perhaps even are.

I would therefore like to express my gratitude to my colleagues who accompanied me on this interesting and effective research journey, including Prof. Dr. Horst Zuse, Prof. Dr. Alain Abran, Harry Sneed, Prof. Dr. Andreas Schmietendorf, Prof. Dr. Cornelius Wille, Anja Fiegler, Prof. Dr. Capers Jones, Prof. Dr. Olga Ormandjieva, Günter Büren, Manfred Bundschuh, Dr. Christof Ebert, Prof. Dr. Jens Heidrich, Dr. Ina Richter, Dr. Robert Neumann, . Dr. Yoshiki Mitani, Carol Dekkers, Dr. Thomas Fehlmann, Dr. Mathias Lother, Dr. Luigi Buglione, Prof. Dr. Juan Gallego, Prof. Dr. Humberto Garcia, Charles Symons, Prof. Dr. Jorga Marx-Gomez, Prof. Dr. Dieter Rombach, Luca Santillo, Prof. Dr. Andre Janus, Prof. Dr. Matthias Hobelsberger, Prof. Dr. Luisa Gonzales, Dr. Martin Kunz, Dr. Rene Braungarten, Prof. Dr. John Munson, Dr. Steffen Mencke, Heike Hegewald, Dr. Ayaz Farooq, Dr. Mathias Riedl, Dr. Cigdem Gencel, Dr. Dmytro Rud and many, many others.

6. References

[Abran 2010] Abran, A.: Software Metrics and Software Metrology. John Wiley & Sons, 2010

[Abran 2006] Abran, A. et al. (Eds.): Applied Software Measurement. Shaker Publ., 2006

[Basili 2002] Basili, V. R. et al.: Lessons learned from 25 years of process improvement: The Rise and Fall of the NASA Sodtware Engineering Laboratory. Proceedings of the ICSE '02: Proceedings of the 24th International Conference on Software Engineering, p. 69 79

[Basili 1984] Basili, V. R.; Perricone, B. T.: Software Errors and Complexity: An Empirical Investigation. Comm. Of the ACM, 27(1984)1, pp. 42 - 52

[Bhowmick 2004] Bhowmick, S. S.; Madria, S. K.; Ng, W. K.: Web Data Management. Srpinger Publ., 2004

[Binder 2000] Binder, B.: Testing Object-Oriented Systems. Addison Wesley Publ., 2000

[Boehm 2007] Boehm, B. W.: Software Engineering. IEEE Computer Society, Los Alamitos, 2007

[Brown 1998] Brown, A. W.; Wallnau, K. C.: The Current State of CBSE. IEEE Software, Sept./Oct. 1998, pp. 37-46

[Brooks 1975] Brooks, F. P.: The Mythical Man-Month – Essays on Software Engineering. Addison Wesley Publ., 1975

[Büren 2011] Büren, G.; Dumke, R. R.; Münch, J.: Metrikon 2011 – Praxis der Softwaremessung. Shaker Publ., Aachen, 2011

[Bundschuh 2008] Bundschuh, M.; Dekkers, C.: The IT Measurement Compendium. Springer Publ., 2008

[Cooper 2010], Cooper, B.: The Prickly Side of Building Clouds. IEEE Internet Computing, Nov./Dec. 2010, pp. 64-67

[Davis 1995] Davis, A. M.: 201 Principles of Software Development. McGraw Hill Publ., 1995

[DeMarco 2009] DeMarco, T.: Software Engineering: An Idea Whose Time Has Come and Gone? IEEE Software, July/August 2009, pp. 95 - 96

[Dijkstra 1968] Dijkstra, E. W.: Go To Statement Considered Harmful. Comm. of the ACM , 11(1968)3, pp. 147-148.

[Dumke 2021] Dumke, R. R.: A Historical Overview of Software Measurement. Software Measurement News, 26(2021)2, pp. 24 – 34

[Dumke 2023] Dumke, R. R.: SML@b News -Software Measurement Apps. Software Measurement News, 28(2023)1, p. 15

[Dumke 2003] Dumke, R.: Software Engineering. Vieweg Publ., Braunschweig, Germany 2003

[Dumke 2011] Dumke, R; Abran. A.: COSMIC Function Points – Theory and Advanced Practices. CRC Publ., Boca Raton, 2011

[Dumke 2008] Dumke, R. R. et al.: Software Process und Product Measurement. Springer Publ., 2008

[Dumke 2003] Dumke, R.; Lother, M.; Wille, Z.; Zbrog, F.: Web Engineering. Pearson Education Publ., Munich 2003

[Dumke 2010] Dumke, R.; Mencke, S.; Wille, C.: Quality Assurance of Agent-Based and Self-Managed Systems. CRC Press, Boca Raton, 2010

[Dumke 2014] Dumke, R., Schmietendorf, A., Seufert, M., Wille, C.: Handbuch der Softwareumfangsmessung und Aufwandschätzung, Logos Verlag, Berlin, 2014

[Dumke 2001] Dumke, R.; Rautenstrauch, C.; Schmietendorf, A.; Scholz, A.: Performance Engineering – State of the Art and Current Trends. Springer Publ., 2001

[Dumke 2002] Dumke, R., Rombach, D.: Software-Messung und -Bewertung, Deutscher Universitätsverlag, 2002

[Ebert 2007] Ebert, C.; Dumke, R.: Software Measurement – Establish, Extract, Evaluate, Execute. Springer Publ., 2007

[Farooq 2009] Farooq, A.: An Evaluation Framework for Software Test Processes. University of Magdeburg, Dept. of Computer Science, PhD, 2009

[Fenton 1993] Fenton N.: System Construction and Analysis – A Mathematical and Logical Framework. McGraw Hill Publ., 1993

[Fenton 1997] Fenton, N. E.; Pfleeger, S. L.: Software Metrics - a rigorous and practical approach. Thompson Publ., 1997

[Gernuchten 2013] Genuchten, M.v.; Hatton, L.: Quantifying Software's Impact. IEEE Computer, pp. 66 - 72

[Hemaspaandra 2002] Hemaspaandra, L. A.; Ogihara, M.: The Complexity Theory Companion. Springer Publ., 2002

[Humphrey 1990] Humphrey, W. S.: Managing the Software Process. Addison-Wesley Pull., 1990

[Humphrey 2000] Humphrey, W. S.: The Personal Software Process: Status and Trends. IEEE Software, Nov/Dec. 2000, pp. 71-75

[ISBSG 2003] Software Project Estimation – *A Workbook for Macro-Estimation of Software Developmet Effort and Duration.* Melbourne, 2003

[Jones 2007] Jones, C. Estimating Software Costs – Bringing Realism to Estimating. McGraw-Hill Publ., 2007

[Jones 2010] Jones, C.: Software Engineering Best Practices – Lessons from Successful Projects in the Top Companies. McGraw-Hill Publ., 2010

[Keyes 2003] Keyes, J.: Software Engineering Handbook. Auerbach Publ., 2003

[Khalidi 2011] Khalidi, Y. A.: Building a Cloud Computing Platform for New Possibilities.IEEE Computer, March 2011, pp. 29-34

[Kitchenham 2007] Kitchenham, B: Empirical Paradigm – The Role of Experiments. In: Basili et al.: Empirical Software Engineering, Springer Publ., pp. 25-32, 2007

[Kitchenham 1996] Kitchenham, B. A.: Software Metrics – Measurement for Software Process Improvement. NCC Blackwell Ltd, 1996

[Kitchenham 1996] Kitchenham, B. A.: Software Metrics – Measurement for Software Process Improvement. NCC Blackwell Ltd, 1996

[Leiss 2007] Leiss, E. L.: A Programmer's Companion to Algorithm Analysis. Chapman & Hall Publ., 2007

[Lorenz 1994] Lorenz, M.; Kidd, J.: Object-Oriented Software Metrics. Prentice Hall Publ., 1994

[McConnel 2006] McConnel, S.: Software Estimation – Demystifying the Black Art. Microsoft Press, 2006

[McGarry 2002] McGarry, J. et al.: Practical Software Measurement – Objective Information for Decision Making. Addison Wesley Publ., 2002

[Menasce 1998] Menasce, D. A.; Almeida, V. A. F.: Capacity Planning For Web Performance - Metrics, Models, & Methods. Prentice Hall, 1998

[Mendes 2006] Mendes, E.; Mosley, N.: Web Engineering. Springer Publ., 2006

[Munson 2003] Munson, J., C.: Software Engineering Measurement. CRC Press Company, Boca Raton London New York, 2003

[Pandian 2004] Pandian, C. R.: Software Metrics – A Guide to Planning, Analysis, and Application. CRC Press Company, 2004

[Putnam 2003] Putnam, L. H.; Myers, W.: Five Core Metrics – The Intelligence Behind Successful Software Management. Dorset House Publishing, New York, 2003

[Richter 2015] Richter, K.; Dumke, R. R.: Modeling, Evaluating, and Predicting IT Human Resorces Performance, CRC Publ., Boca Raton, 2015

[Rud 2009] Rud. D.: Performancebewertung und -sicherung von orchestrierten Serviceangeboten. Kovac Publ., Hamburg, 2009

[Schmietendorf 2007] Schmietendorf, A.: Eine strategische Vorgehensweise zur erfolgreichen SOA-Implementiereung. Shaker Publ., Aachen, 2007

[Schmietendorf 2002] Schmietendorf, A.; Dimitrov, E.; Dumke, R.: Enterprise JavaBeans. Mitp Publ., 2002

[Shneiderman 1980] Shneiderman, B.: Software Psychology. Winthrop Publ., Massachussetts, 1980

[Singpurwalla 1999] Singpurwalla, N. D.; Wilson, S. P.: Statistical Methods in Software Engineering. Springer Publ., 1999

[Sneed 2010] Sneed, H. et al.: *Software in Zahlen*. Hanser-Verlag München, 2010

[Sommerville 2010] Sommerville, I., Software Engineering. Addison-Wesley, 2010

[Whitmire 1997] Whitmire, S.A.: Object Oriented Design Measurement. John Wiley & Sons, 1997

[Wille 2005] Wille, C.: Software Agent Measurement Framework. Shaker Publ., Aachen, 2005

[Zuse 1998] Zuse, H.: A Framework of Software Measurement. De Gruyter Publ., Berlin, 1998