Int J Software Informatics, Volume 5, Issue 3 (2011), pp. 525–534
International Journal of Software and Informatics, ISSN 1673-7288

E-mail: ijsi@iscas.ac.cn
http://www.ijsi.org
Tel: +86-10-62661040

Empirical Software Engineering Models: Can They Become the Equivalent of Physical Laws in Traditional Engineering?

Dieter Rombach

(University of Kaiserslautern & Fraunhofer IESE, 67663 Kaiserslautern, Germany)

Abstract Traditional engineering disciplines such as mechanical and electrical engineering are guided by physical laws. They provide the constraints for acceptable engineering solutions by enforcing regularity and thereby limiting complexity. Violations of physical laws can be experienced instantly in the lab. Software engineering is not constrained by physical laws. Consequently, we often create software artifacts which are too complex to be understood, tested or maintained. As too complex software solutions may even work initially, we are tempted to believe that no laws apply. We only learn about the violation of some form of "cognitive laws" late during development or during maintenance, when too high complexity inflicts follow-up defects or increases maintenance costs. Initial work by Barry Boehm (e.g., CoCoMo) aimed at predicting and controlling software project costs based on estimated software size. Through innovative life cycle process models (e.g., Spiral model) Barry Boehm also provided the basis for incremental risk evaluation and adjustment of such predictions. The proposal in this paper is to work towards a scientific basis for software engineering by capturing more such time-lagging dependencies among software artifacts in the form of empirical models and thereby making developers aware of so-called "cognitive laws" that must be adhered to. This paper attempts to answer the questions why we need software engineering laws and how they could look like, how we have to organize our discipline in order to build up software engineering laws, what such laws already exist and how we could develop further laws, how such laws could contribute to the maturing of science and engineering of software in the future, and what the remaining challenges are for teaching, research, and practice in the future.

Key words: engineering & physical laws; software engineering & cognitive laws; observations, laws & theories; empirical methods; science & engineering

Rombach D. Empirical software engineering models: Can they become the equivalent of physical laws in traditional engineering?. *Int J Software Informatics*, **Vol.5, No.3 (2011): 525–534. http://www.ijsi.org/1673-7288/5/i94.htm**

1 Motivation & Introduction

Why do we need a scientific basis in software engineering based on some form of "laws", and how could these laws look like? Physical laws provide the scientific basis for scaling up engineering in traditional engineering disciplines. For example, all microelectronics is based on laws in semiconductor physics. These physical laws impose

Corresponding author: Dieter Rombach, Email: Dieter.Rombach@iese.fraunhofer.de
Paper received on 2011-03-14; Revised paper received on 2011-03-28; Paper accepted on 2011-03-29; Published online 2011-04-07.

constraints for acceptable solutions and require high degrees of regularity to manage scale-up complexity. Violations of physical laws can be experienced instantly in the laboratory. Software engineering is not constrained by physical laws. Consequently, we often create software artifacts which are too complex to be easily understood, tested or maintained. As too complex software solutions may even work initially, we are often tempted to believe that no laws exist in the software domain. We only experience the violation of some form of "cognitive laws" late during development or during maintenance, when too high complexity inflicts follow-up defects or increasing maintenance costs. Problems are the time lag until violations of software laws occur, and the cognitive nature of software laws. The time lag could mean that a too complex design could lead to an initially functioning software system, but later during maintenance changes result in follow-up defects resulting from the maintenance of a too complex – and thereby badly understood – software system. In the context of this paper we refer to "cognitive laws" as qualitatively stable models (see chapter 3.1) relating characteristics of different software artifacts. They typically are related to cognitive capabilities of software developers to comprehend software artifacts. The definition is NOT explicitly related to definitions in the field of Cognitive Science. For example, initial work by Barry Boehm (e.g., CoCoMo) related estimated size of a software product with the costs for its development; this model is widely being used today for software project cost prediction and controlling[6]. Furthermore, through innovative life cycle process models (e.g., Spiral model) Barry Boehm provided the basis for incremental evaluation and adjustment of such predictions[7]. The empirical nature of our field implies that our "cognitive laws" can only be detected empirically via time-consuming studies and have to be adapted to different project contexts. The proposal in this paper is to work towards a scientific basis for software engineering by capturing these long-term dependencies between construction and behavior as empirical process-product models and thereby making developers aware of so-called cognitive laws that must be adhered to. This paper motivates the need for cognitive laws as software engineering equivalents to physical laws in traditional engineering, introduces existing examples, and suggests a community effort to advance the states of research and practice.

1.1 Engineering

Most traditional engineering disciplines such as mechanical, electrical or civil engineering depend on physical laws.

1.1.1 Physical laws

Laws from semiconductor physics guide all microelectronics. They define the packaging density of chips. High density of packaging is achieved by extreme regularity of computer hardware. Such law-enforced regularity has been the pre-requisite for scaling the engineering of computer hardware.

1.1.2 Benefits

The benefits are that we are limited in our solution space. For example, in order to pack large numbers of functionality on a chip, rules about distances between connections need to be adhered to, and regular patterns are a key to scale up. As a

result, solutions look uniform; there is not much space for unnecessary creativity.

Education in engineering to adhere to physical laws is easy, as violations lead to immediate failures and can be easily experienced in lab projects. This early experience feedback leads to unquestioned acceptance of regular and complexity-reducing construction principles.

1.2 Software engineering – cognitive laws

Software engineering is not dependent on physical laws. Often it seems we are not dependent on any laws. However, the creation of any complex human-made artifact – especially such immensely complex artifacts as software – must be easily understood, tested, and maintained. So what are the boundaries or "laws" that constrain our ability to understand, construct, test or maintain? Concepts like "design for testability" are intended to guide design decisions based on their potential impact on testability. Such concepts need to be captured empirically in quantitative terms, and then formalized as "software engineering laws". These software engineering laws are based on cognitive abilities of individual developers and, therefore, referred to as "cognitive laws". They describe limitations of the human ability to intellectually comprehend software artifacts. Further examples include relationship between design complexity and the ability to test or maintain well. Every practitioner has experienced that above a certain design complexity threshold it becomes hard to test systematically. We all know the consequences of not adhering to certain complexity limitations in the form of residual software defects in operational software as a result of inadequate comprehensibility or testability, or the inability to maintain software (presumably its premier advantage over hardware) without introducing new defects or utilizing unacceptably high resources.

So why do we not adhere to such cognitive laws as engineers adhere to physical laws (see complexity of a typical software system). The answer is multifold: (1) These laws can only be experienced with a time lag, and most phase-based life-cycle models do not provide feedback e.g. from testing to design, (2) they can only be determined empirically and are different for different humans, and (3) computer science has not yet established a broad basis of such laws. Whereas an engineering student could experience the violation of physical laws immediately in the lab, a software engineering student may get a software system despite violation of complexity laws to run, and thereby establish the illusion that there are no boundaries for the construction of software. Very rarely will he/she experience the impact of such violations during maintenance. Physical laws exist and are universally applicable. The establishment of cognitive software engineering laws requires time-consuming empirical effort (even if laws exist in one environment, due to the personalization of cognition they would have to be re-validated empirically for a new environment). Many software engineering organizations shy away from this effort. Computer Science has been mostly created from mathematics, and keeps following mathematical paradigms of optimal answers to problems. Science requires that results must be challengeable. How can we claim to be a science if human-based techniques such as testing are not augmented with some impact statement (e.g., test technique T can be applied with a certain effectiveness Q provided the complexity is not exceeding a threshold C). Such empirical dependencies can be established by researchers via controlled lab experiments, or by practitioners

via field case studies.

1.2.1 Cognitive laws

The general form of empirical models

$$Q/P/T(A1) == f(A2, C) \tag{1}$$

where $A1$ and $A2$ are models of software artifacts – both products and processes, $Q/P/T$ is any aspect of quality (Q), cost and productivity (P), or time (T) of artifact $A1$, f is the function capturing the relationship, C is the context (e.g., developers' experience, project size) for which the relationship holds, and "==" denotes that the relationship is empirical with some uncertainty (e.g., $+/-5\%$).

That means that we have four kinds of models:

- Product-product models: Examples include the relationship between complexity of a software design and the quality (number of residual defects) of the final software[4].
- Product-process models: Examples include the well-known CoCoMo model from Barry Boehm[6] where $A1$ is the effort distribution model of effort over lifecycle phases, $A2$ is the size model (measured in terms of #LoC), P is Effort, C is described in terms of 14 impact factors, f is "a * Size (power b)", and the uncertainty is specific to every organization. Other examples exist[13].
- Process-product models: Examples include models describing the effectiveness of methods (e.g., inspections, testing) on cost and quality[2,15]). These models are especially important to choose the methods and tools appropriate for a set of project goals and context characteristics during project planning. For example the effects of a testing technique may be described as follows:
  - Testing technique T identifies 80% of all defects ($+/-5\%$), if the code complexity is below a threshold C and testers experience is high
  - Testing technique T identifies 65% of all defects ($+/-10\%$), if the code complexity is above a threshold C and testers experience is high
  - Testing technique T identifies 50% of all defects ($+/-20\%$), if the code complexity is above a threshold C and testers experience is average or low
  - Etc.
- Process-process models: Examples include effort distribution models that relate the relative project effort across different phases of development.

We call any equation of form (1) a "law" if for all relevant contexts function f shows the same qualitatively stable effect (e.g., positive or negative). Such a function could on the one hand be the result of a controlled test experiment varying software complexity and tester experience. In this case, the significance of any cause-effect relationship would be high, but the scaleability to practice would still be questionable. It could on the other hand be the result of observations in an industrial environment over a number of projects with varying complex software and varying tester experience. In this case the scaleability would be high, but there would a residual risk regarding cause-effect and therefore repeatability in future projects.

1.2.2 Challenges

Cognitive laws are based on empirical evidence. Four specific challenges include

- The choice of study (controlled experiements, case studies)
- The maturing of characterizing models to predicting models
- The accumulation of observations from individual studies to "laws" and theories
- The adaptation / generalization of observations, laws and theories to other / wider contexts[14]

Model-based hypotheses must be tested by a series of controlled experiments and / or case studies. From a theoretical perspective it is generally impossible to test all possible combinations of context. Therefore, such laws will never be established purely statistically. From a practical perspective, it is sufficient to test all critical contexts and then generalize based on general knowledge in the field. The same procedure is applied in medical studies where the same constraints apply.

In general, we have to accept the fact that software is designed and not manufactured. The effects of design processes depend highly on context. The challenges towards establishing a science of software engineering include the creation of empirical observations, the aggregation of empirical observations into laws, and ultimately the establishment of theories. Empirical observations are based on individual controlled experiments or case studies. They represent one data point – valid for a limited context (e.g., one project in the case of case studies) and with limited statistical significance. Empirical laws represent generalizations of aggregations of empirical observations with a common qualitative effect trend – valid for a certain context and tested enough to establish community trust. Examples of empirical laws are "use of systematic reading of requirements reduces the number of defects and rework effort"[8]. A large number of experiments and case studies had been performed – all showing a positive trend despite quantitative differences due to context differences. The community seems to have accepted the existing evidence due to its representative coverage of different types of software and contexts. Such evidence should be declared a law – implying that practitioners have to apply it or take the responsibility for possible negative consequences.

Especially the accumulation of obersvations to laws and theories will be discussed further in section 3.1.

2 Software Engineering as a Discipline

How do we have to structure our discipline of software engineering in order to include the development and use of empirical models into research and practice (better: how do we advance towards a true engineering discipline)? What is the state of the art and practice in our discipline? Software engineering – like any other engineering discipline – must address questions related to modeling (e.g., what informal and formal notations are appropriate to model software systems?), system technology (what principles for structuring complex software systems are appropriate?), and process technology (what processes must be followed from capturing requirements to delivery and maintenance?). In addition, empirical studies must be performed in order to establish cognitive laws for all the above.

2.1 Structure of the discipline

An engineering-style structure of the Software Discipline taking into consideration its human-based cognitive nature is described in Fig.1.

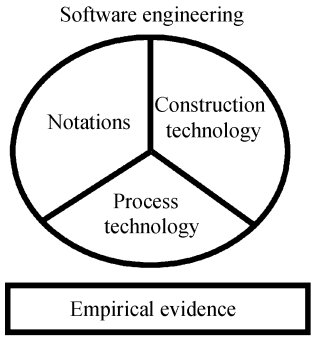

Figure 1. Structure of SE discipline

It is composed of 4 major sub-areas: Notations, construction technology, process technology, and empirical evidence. Examples for notations include programming languages, design and requirements languages, or documentation standards. Examples for construction technology include topics such as architectures, software product lines, software reuse, or general modularization concepts. Process technology addresses life-cycle models and project management practices – everything needed to engineer a software system from initial needs. Finally, due the nature of software and software engineering, empirical evidence is needed to establish cognitive laws. The methods and tools to perform such empirical studies exist[1,3,5]. It can be observed that the majority of research in the past is related to notation and system technology. Limited research has been done in process technology and empirical studies.

In other technical disciplines we have a clear separation of science versus physics. For example in the hardware world, physics represent science, and electrical and mechanical engineering represent engineering. In the software domain, both science and engine erring are typically represented by by computer science. We need to learn to also distinguish between science (including the creation of cognitive laws) and its application in engineering.

2.2 Characteristics of the discipline

Engineering can be described as all activities efficiently developing and / or producing human-based artifacts according to plan and with certifiable quality. This requires the existence of explicit models, their use for planning, and their use for early defect prevention and detection. Finally, management of complexity supported by methods & tools – like in other engineering disciplines – is the key to scaling up engineering.

2.2.1 Explicit models

This includes models of systems and system aspects at all levels of abstraction – including requirements, architecture & design, and coding. Notations can be formal (e.g., programming languages such as JAVA, or requirements languages such as

Sequence-Based Specification) or informal (e.g., structured English for requirements) depending on the quality of service guarantees required. Furthermore, models are required for process (e.g., life-cycle models) or qualities (e.g., defect detection distributions in percentages across life-cycle phases). The status is that especially for process and qualities only few models exist. In the context of model-based development the importance of sound product models becomes more evident. A more systematic characterization of models will be listed in chapter 4.

2.2.2 Planning & quality assurance

One important aspect of engineering is the ability to plan and assure adherence to plan throughout projects. Engineering-stely planning and quality assurance must be based on models. Existing prediction models include cost and effort predictions based on anticipated system size (e.g., CoCoMo by Barry Boehm) or requirements size (e.g., Function Points). Both of these prediction models can predict cost and time based globally (per project) or locally (per project phase). In the latter case, continuing adherence can be measured. Barry Boehm's CoCoMo model established the first "cognitive law" of type (1) as he established a qualitatively stable relationship between system size and effort, but showed quantitative differences based on context variables. Other planning models include the prediction of residual software defects (so-called reliability models) or defect detection models across development phases based on empirical observations. All of these models allow prediction and quality assurance. Barry Boehm with his risk-based life-cycle models provided the ability to learn and improve within projects.

2.2.3 Early focus on prevention & detection of defects

Again, Barry Boehm provided the arguments why defect prevention and early defect detection – like in other engineering disciplines – pays off in terms of quality and rework cost reduction. The general law presented by Barry Boehm is that the cost of rework for a given defect increases by a factor of 10 per development phase it remains in the system. So if the cost for removing a requirements defect during the requirements is "1", then its removal or rework cost during design is "10". As a result of this law, systematic inspection or review techniques (based on systematic reading or analysis techniques for software artifacts such as perspective-based reading) have been developed and in multiple empirical studies have been demonstrated to always reduce rework in practice.

2.2.4 Complexity management – supported by methods & tools

The central issue for scaling up engineering is complexity management. The basic principles for managing the complexity of software systems are long known[12]. The challenges today are the development of model-based software development approaches and their support by tools. The main ideas are to employ either model-based generation or use of patterns[9]. Many empirical studies within the software comprehensibility and maintainability communities show that individual thresholds of complexity represent barriers for good development or maintenance performance.

3 Empirical Models

What is the nature of our software engineering laws? How do we mature empirical observations into empirical laws and theories? What is the state-of-the-art and – practice in using software engineering laws?

3.1 Empirical observations, laws, and theories

Empirical models have associated levels of significance or trust which is related to the frequency of observation and the uniformity of result repetition. Single observations are relevant to one single project; no guarantee is given that it can be repeated in future projects. The term "law" refers to a set of observations with representative coverage of a project domain (often attested by experts in technology and domain) and trend-wise (qualitatively stable) identical results. For example, in the area of requirements inspections we have enough replicated studies[11] that cover all relevant variations of software types, experience of inspectors, etc., and all show a positive effect wrt. effectiveness and efficiency. Although the quantitative improvements differ (from 5% to 50% depending on organizational maturity), we can formulate a cognitive law saying "Use of systematic requirements inspections saves effort and reduces the number of requirements defects!". A law becomes a theory, if all independent context factors are known and, therefore, good predictions for future projects become feasible. A detailed discussion regarding the definition of observations, laws, and theories is contained in Ref.[8].

3.2 State of the practice

In practice most organizations apply processes, techniques and methods based on "we always did it this way" principle, or based on prominent buzz words (e.g., agile) without even knowing whether these approaches benefit the organization to what degree! The consequences are (Ref: Chaos report) many project failures, especially a wide variation of project outcomes (up to 100% miss of quality, productivity, or time targets). This should not be surprising if approaches are used across projects with differing characteristics. Good engineering should focus on repeating project results under varying project characteristics. This must imply (often slight) change of process. A German automotive supply company has demonstrated that adequate variation of inspections techniques (ad hoc for experiences inspectors; check list based or perspective based inspections for less experiences inspectors) has resulted in and guaranteed similar projects results in terms of defect detection removal effectiveness.

3.3 State of the art

There exists a large body of observations, laws, and theories (e.g., [8]) – many observations, fewer laws, hardly any theories. These empirical observations, laws, and theories should be known to practitioners (and compliance required as good practices, or reasons given for waivers), used as reference for professional behavior in legal disputes, taught to students early on, and complemented by researchers and practitioners.

4 Benefits for our Discipline

How does this empirical or "cognitive" law-based view of software engineering benefit science and engineering of software (better: how far are we on the path towards a true science-based engineering discipline)?

4.1 Science

Science requires that software engineering results can be challenged. Therefore, religious claims about new processes (e.g., testing) to be better do not qualify as scientific results. Scientific results in the form of (1) do qualify. Therefore, all software engineering researchers must provide such evidence – themselves or in cooperation with others. Although the evidence captured in Ref.[8] provides a good starting point, it is the responsibility of each software engineering researcher to contribute more evidence in his/her respective area of expertise.

4.2 Engineering

Engineering requires that software methods and tools for a project can be chosen based on project goals and project context. In order to do so, we need a pool of best practice methods and tools with empirical effectiveness statements of the form of equation (1). In most organizations, choices are made on "we did it always this way" or "subjective claims". To no surprise this results in projects being off target by more than 100%. It should be the responsibility of SEPGs in companies to compile best practice "laws" of the form (1) so that project choices can be made engineering-style.

5 Future Challenges for the SE Community

What remains to be done to continue this path in the future? We certainly need to rethink our views regarding education, science, and engineering. All of us need to acknowledge the largely empirical or cognitive nature of our discipline and act accordingly.

5.1 Education

We need to change our education (especially at the freshmen level) from trial-and-error programming to law-based programming. This requires that we initially allow freshmen to self-experience compliance as well as non-compliance with software engineering laws by means of understanding and changing good and bad programs. Equipped with such self-experience they will be able to appreciate and accept these software engineering laws.

5.2 Science

No science without challengeable results. Results such as "I have defined a useful new testing technique" remind of religious claims and can never be challenged or falsified. Results such as "I have defined a new testing technique which yielded in an experiment (detailed description of experimental design, data analysis, and interpretation) 30% higher defect detection effectiveness" can be challenged by repetition of the specified experiment[11]. We definitely need more experiments including replications[11], need to apply empirical studies to large-scale development processes[14], and compose

individual observations to laws and theories[8].

5.3 Engineering

Practitioners need to accept the existence of best practice laws[1]. They also need to understand that these laws need to be adapted to their specific project contexts[10]. It has to become professional ethics to know and apply those laws. In case of non-compliance personal responsibility for potential failures must be accepted.

6 Summary

Barry Boehm has been the first software engineering researcher who has mentioned the need for something like cognitive software engineering laws. In his CoCoMo Model he has captured an equation for cost and time of type (1). He has demonstrated that there is not predictor that fits all contexts, but that the context must be specified explicitly. He has also defined risk-based life-cycle models which – through iteration and evaluation of risks – feed back experiences from later phases to decision makers of earlier phases. We have started to capture existing observations and laws[1]. However, this can only be the beginning. We as a community must strengthen our activities to build a sufficient body of knowledge regarding key software engineering practices. This would begin to define a science of software engineering which will enable true engineering of software, and prediction of project outcomes with acceptable accuracy.

References

[1] Basili VR, Selby RW, Hutchins DH. Experimentation in Software Engineering. IEEE Trans. on Software Engineering, 1986, 12(7): 733–743.

[2] Basili VR, Selby RW. Comparing the effectiveness of software testing strategies. IEEE Trans. on Software Engineering, 1987, 13(12): 1278–1296.

[3] Basili VR, Caldiera G, Rombach HD. Goal Question Metric Paradigm. In: Marciniak JJ ed. Encyclopedia of Software Engineering. New York: Wiley, 1994. 469–476.

[4] Basili VR, Briand LC, Welo WL. A Validation of object-oriented design metrics as quality indicators. IEEE Trans. on Software Engineering, 1996, 22(10): 751–761.

[5] Basili VR, Green S, Laitenberger O, Lanubile F, Shull F, Soerumgard S, Zelkowitz MV. Empirical Software Engineering, 1996, 1(2): 133–164.

[6] Boehm BW. Software Engineering Economics. Englewood Cliffs, NJ: Prentice-Hall, 1981.

[7] Boehm BW. A spiral model of software development and enhancement. IEEE Computer, 1988, 21(5): 61–72.

[8] Endres A, Rombach HD. A Handbook for Software and Systems Engineering. Harlow, UK: Pearson, 2003.

[9] Gamma E, Helm R, Johnson R, Vlissides J. Design patterns: Elements of Reusable Object-Oriented Software. Reading, MA: Addison-Wesley, 1995.

[10] Humphrey WS. Using a defined and measured personal software process. IEEE Software, 1996, 13(3): 77–88.

[11] Lott CM, Rombach HD. Repeatable software engineering experiments for comparing defect-detection techniques. Empirical Software Engineering, 1996, 1(3): 241–277.

[12] Parnas DL. On the criteria to be used in decomposing systems into modules. Communications of the ACM, 1972, 15(12): 1053–1058.

[13] Rombach HD. A controlled experiment on the impact of software structure on maintainability. IEEE Trans. on Software Engineering, 1987, 13(3): 344–354.

[14] Selby RW, Basili VR, Baker FT. Cleanroom software development: An empirical investigation. IEEE Trans. on Software Engineering, 1987, 13(9): 1027–1037.

[15] Travassos GH, Shull F, Fredericks M, Basili VR. Detecting defects in object-oriented designs: Using reading techniques to increase software quality. In: Proc. Conf. on Object-Oriented Programming, Languages, and Applications (OOPSLA), Denver, CO, 1999.