About Theory in Software Development

JAVIER ANDRADE¹, JUAN ARES¹, RAFAEL GARCÍA¹, MARÍA-AURORA MARTÍNEZ², SANTIAGO RODRÍGUEZ¹, SONIA SUÁREZ¹
¹Information and Communication Technologies
University of A Coruña
Campus de Elviña, s/n. 15071. A Coruña
SPAIN
{jag, juanar, rafael, santi, ssuarez@udc.es}
²UDIMA-Madrid Open University
Camino de la Fonda, 20. 28400. Collado-Villalba, Madrid
SPAIN
mariaaurora.martinez@udima.es

Abstract: - Despite being two very closely linked things, hardware and software have evolved very differently. In actual fact, while hardware evolution has been exponential, software has evolved linearly. One of the fundamental reasons for this mismatch between hardware development and software development is that hardware development is supported by engineering based on science whose scientific theory is very well established, whereas software development is not. The goal of the present work is the analysis of, not only software historic development, but also the current theoretical approaches for software development. The results achieved will be used for proposing new research lines that might help this discipline to advance in the scientific field. It should be then possible to establish the foundations that might enable software advance similarly as hardware does.

Key-Words: - Software development, theory, science, principle, holon, informon

1 Introduction: Hardware versus Software

In 1947 John Bardeen and Walter Houser Brattain invented the transistor. Only three years later, Jack Kilby, of Texas Instruments, patented the integrated circuit or “chip”. This building block is one of the things that really most marked the history of hardware. In actual fact, the microprocessor, which is the chip responsible for controlling all the other chips in a computer and carrying out all key operations, is unquestionably the most important component of today’s computers. Intel presented the first microprocessor 1971. Microprocessors have evolved at a really giddy rate since then. In fact, we find that technological evolution is amazingly exponential [1]. As regards this evolution, Kurzweil noted that two complementary laws came into play over time. The first, known as the law of increasing chaos, gives account of the empirical fact that time slows down as chaos increases. In other words, the interval between two relevant events grows with time. The second law is called the law of increasing returns. This law account of the fact that time slows down exponentially as order grows exponentially. In other words, the interval between two important, relevant and significant events decreases with time.

Moore’s law is a well-known corollary and paradigmatic example of this law. It was stated by Forrester in 1987 [2] as follows: computing power doubles every 18-24 months at the same unit cost and there is a trend towards a shorter time period.

It will continue until the limits of the physical laws governing current chips [3], that is, conventional electronics, are reached.

To give a clear idea of the extraordinary real increase of computing power, let us recall that it has grown by approximately eleven orders of magnitude from 1950 to the present day. In other words, it has multiplied by 10¹¹.

And giving account of this incredible evolution is the aforesaid Moore’s law and its different interpretations (see, e.g., [4]). Moore’s law also has a point of inflection, which is defined by the underlying technology (the transistor) and the point-one barrier. Having reached this point, a new technology will be needed to manufacture microchip components. In this respect, the proposal of the
Nobel prize-winning Heinrich Rohrer is nanotechnology [5].

The limits of this hardware evolution, as pointed out by Seth Lloyd [6], are determined by several natural constants: the speed of light, the quantum scale or the reduced Planck constant, the gravitational constant and Boltzmann’s constant. This fact occurs because computers are physical systems and are, therefore, subject to the laws of physics, which are what ultimately dictate what they can or cannot do, at least in theory.

Comparing the progress of software with the evolution of hardware presented above, we come across two clearly distinct facts. First, it is plain that the development of hardware is in depth, that is, there are few companies (Intel, Motorola, etc.) engaged in developing microprocessor technology, whereas this effort is carried out in breadth in the case of software, that is, there are many companies putting a lot of very varied products on the market.

Secondly, and this is the fact with which we are most concerned here, the progress of software is very far from being exponential. Indeed, considering all the software available world-wide, both basic (operating systems, compilers, etc.) and applications software, its evolution is linear and is, by no means, continuously increasing in terms of features and reliability. By way of an example, it is noteworthy that, after fifty years of progress [7], contemporary applications software is still regularly characterized by reduced functionality, reduced scope and questionable quality [8]. Thus, for example, computing history is awash with software failures and their adverse impacts on individuals, organizations and societal infrastructure [9] (cf. e.g. [10, 11, 12, 13, and 14]).

The evolution of applications software is, of course, mainly subject to the actual evolution of programming languages (basic software) used in its development. The Mother Tongues of Computer Languages poster, originally published in Wired Magazine and available on-line at [15], shows the tree that reflects the evolution of programming languages. This poster portrays a genuine Tower of Babel, tracing the gradual evolution of the languages used for programming. The evolution of these languages and of software generally, because of its programming languages dependence, can be seen primarily as the step from “how” to “what” [16].

This slow, meandering and protracted evolution of software contrasts with the rapid, straight and efficient evolution of hardware. The obvious question is, of course, “Why is this so?” There is, evidently, unlikely to be just one factor, but there can be no doubt that one of the fundamental reasons for this mismatch between hardware development and software development is that hardware development is supported by engineering based on science—e.g. physics—, whose scientific theory is very well established, whereas software development is not. In the case of software development, there is actually a notorious absence of theory (cf. e.g., [17]) to the point that, except on very rare occasions, there are not even any commonly accepted general laws. Because of this missing theory, software development is, at best, an immature science. Others think that it is not a science but an engineering discipline [18]. And, at worst, it is an immature engineering discipline [19]. We view software development in its present state as an empirical practice that takes from sciences and engineering what it needs but is founded on neither.

The non-existence of a software development theory contrasts with the everyday reality of other fields of science (cf. e.g. [20]), where there are indeed established scientific theories on which to found studies, compare experiments or base the respective (mature) engineering disciplines. With respect to this point, it is worth mentioning that Kyle Eischen’s comparison [21] of software development and other engineering disciplines led him to affirm that basic patterns (software manufacturing, software industrialization, and software assembly lines) common to all industries are far from being dominant in the software world. Indeed, Michael Jackson stated in his latest book on software engineering [7] that, despite fifty years of progress, software engineering is still in its infancy and is immature. According to Christof Ebert [19], the reason for software engineering’s immaturity is precisely that it has no scientific theoretical foundations.

Ultimately, the underlying question is, “Should software development continue to be the craft of developing an empirical practice or should it become engineering based on scientific theory?” Evidently, the answer is, in our opinion and that of many others, such as those referenced above, that it should be engineering supported by scientific theory.

This paper is organized as follows. Section 2 describes the state of the art of the theories for software development in contrast to other science
fields. Section 3 present the work lines proposed for minimizing the evolutionary gap between hardware and software development. Lastly, section 4 presents the most noteworthy conclusions.

2 Theories

According to the United States National Academy of Sciences [22], a theory is defined in science as “a comprehensive explanation of an important feature of nature supported by facts gathered over time. Theories also allow scientists to make predictions about as yet unobserved phenomena”.

Table 1 shows a group of well-known scientific theories. Each of them supports a scientific area, so any research on a given area should be subject to that theory. These theories explain one specific aspect of reality, according to what is known from it up to the present time, and they enable predicting future events. They are also formulated from few principles/postulates and with few basic concepts.

<table>
<thead>
<tr>
<th>Theory</th>
<th>Author</th>
<th>Year</th>
<th>Principles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hydrostatics</td>
<td>Archimedes</td>
<td>287-212</td>
<td>2</td>
</tr>
<tr>
<td>Mechanics</td>
<td>Newton</td>
<td>1665-86</td>
<td>4</td>
</tr>
<tr>
<td>Atomic Chemistry</td>
<td>Dalton, Thomson, Rutherford, Bohr, Pauli</td>
<td>1803, 1897, 1938, 1912, 1924</td>
<td>3</td>
</tr>
<tr>
<td>Evolution</td>
<td>Spencer, Wallace, Darwin</td>
<td>1952, 1858, 1859</td>
<td>2</td>
</tr>
<tr>
<td>Electromagnetism</td>
<td>Maxwell, Faraday, Hertz, Oested</td>
<td>1873, 1821-31, 1887, 1820</td>
<td>4</td>
</tr>
<tr>
<td>Thermodynamics</td>
<td>Helmholtz, Clausius, Boltzmann, Carnot, Kelvin, Nernst</td>
<td>1850, 1868, 1847, 1829, 1852, 1905</td>
<td>2</td>
</tr>
<tr>
<td>Relativity</td>
<td>Einstein</td>
<td>1905, 1915</td>
<td>3</td>
</tr>
<tr>
<td>Quantum Mechanics</td>
<td>Planck, Einstein, Bohr, Schrödinger, Heisenberg, Dirac, Born, Jordan</td>
<td>1900, 1905, 1926, 1925</td>
<td>3</td>
</tr>
</tbody>
</table>

Tabla I. Scientific theories

Other disciplines like pedagogy, psychology, medicine, etc. are also based on theories.

Kant Lewin said: “There is nothing so practical as a good theory” [23]. Without theories, we would approach phenomena in a piece-meal, descriptive, and post hoc manner. Importantly, the function of a theory is to guide researchers in formulating research questions and hypotheses and in selecting or creating the appropriate methods and strategies for testing those hypotheses. However, without knowing what goals they want to attain (i.e., what theory to test), researchers would easily drown in the sea of potential methodologies not guided by theories. In effect, a theory is to a research method as a horse is to a carriage. Putting the carriage in front of the horse would never work [24].

Like these, there are many other arguments for the use of theories. Moreover, theories provide common frameworks that allow the structuring of knowledge in a precise and concise way and that facilitate the communication of ideas and knowledge. Their level of abstraction allows the generalization of knowledge independently of space and time. These arguments have been broadly defended by the software development community [25, 26, 27, 28, 29, and 30], stating that a theory provides explanations and understanding in terms of basic concepts and underlying mechanisms, which constitute an important counterpart to knowledge of passing trends and their manifestations [31].

Thus, it seems clear than theories are of potential use to researchers and professionals in the field of software development. However, the usefulness of theories for software development is at this moment a discussion issue, and the current use of theories in this discipline is not well known.

Discussions regarding issues of theory in software engineering must be founded not only on an understanding of what a theory is and how it can be useful, but also on knowledge of the actual use of theories [17].

Even so, callings for a “good theory” in software development [31] and for the development of an “own” theory for software development [32] are continuous.

The most complete study related to software development theories was performed by Gregor [30], who considers that there are 5 types of theories.

Type I/Analytical theories answer the question “what is?” Some examples are the work frameworks, the classification diagrams or the taxonomies.
Type II/Explanative theories describe how and why the phenomena occur. The case studies and the surveys correspond to this type. Type III/Predictive theories say how it will be but not why. The Moore law and estimation models as COCOMO are some examples.

Type IV/Explanatory and Predictive theories say what is, how, why, when and what will be. Some examples are the Shannon’ Information Theory or the TAM model.

Type V/Design and Action theories say how something should be done. The methodologies and the guides are included in this type.

As it can be noticed, the theory concept of Gregor is very wide so that anything in software development may be a theory. For instance, the classifications, typologies and taxonomies are Type I theories, however, nobody would consider the periodic table of elements of Chemistry as a theory. Therefore, the conceptual laxity regarding the theory concept in software development seems quite obvious. In the author’s own words: “the word theory will be used here rather broadly to encompass what might be termed elsewhere conjectures, models, frameworks, or body of knowledge” [30]. However, these terms are not synonyms and they are not used that way in any scientific discipline.

This fact appears even clearer in the study carried out by Hannay at al. [17]. These authors analyzed 103 articles with experiments in order to detect references to their subjacent theories and afterwards classifying the theories according the 5 types proposed by Gregor [30]. The requirements for considering an element as theory were the mere mention of the terms “theory”, “model” or any of their grammar derivations; failing these, the identification of constructions and propositions, together with explanations of effect-cause relations were also considered. Once more, the theory concept remains fully open and, again, anything can be theory in software development.

The amount of theories detected by these authors contrasts with the reality in other fields, where there are not a big number of theories but there is a theory to explain and predict a type of phenomena. When a new theory appears, it has to compete with the previous one (in some rare and especial cases they can coexist) to determine which of them explains and predicts better the phenomena. Moreover, these theories reflect the entire discipline and they are not isolated knowledge islands. Also, as table 1 shows, they propose a short set of concepts and principles to represent the phenomena in the discipline.

Along this line, Zendler [33] proposes what intends to be a preliminary theory for software engineering. This theory comprises, according its own nomenclature, 3 fundamental hypotheses, 6 central hypotheses and 4 elemental hypotheses. As a matter of fact, these hypotheses are a group of conclusions that the author reached after analyzing the experiments, which is not a synonymous of theory. The conclusions are considered as hypotheses when they are different concepts; besides, the conclusions are not general, as they regards specific techniques for analysis, design, implementation, testing, maintenance, quality assurance and reuse, leaving aside many of the aspect of software development. With this formulation, the theory could never be general. It doesn’t cover many other aspects of software development (e.g. conceptualization, metrics, estimation, planning and formal methods of evaluation) and it doesn’t covers any new technique in the activities considered by the theory.

In turn, a scientific theory should be general, meaning it should cover a complete science field. The only research known that follows this objective is the work of Alonso et al. [34], who propose the basic elements of a theory for software development. These elements are the holon and the informon, together with the holarchy.

These authors define, according Koestler [35], holon as a processing element that simultaneously has autonomous and not self-sufficient behavior. An informon is defined as a basic information element that enables the holon to make decisions and also execute suitable actions. A holarchy is a system of self-regulated holons that cooperate for reaching a global objective. Also a holarchy may be considered as a unique holon by means of the abstraction of its internal composition.

3 Work lines
The approach of Alonso et al. [34] is a first step for the elaboration of a theory for software development that, once formulated, will be the basis for proposing and contrasting experiments. However, that proposal should be improved with practical/instrumental usefulness.

A theory should have instrumental usefulness and being practicable for solving problems in its domain. A theory can not be stuck in abstract concepts without offering an operative specification for them that might the basis for organizing the elements of the domain. However good a theory might be, if it is operatively unmanageable, it would be almost useless.
Therefore, the following step should be solving such problem by providing the theory elements (holons, holarchies, informons) with an instrumental specification; that is to say, practice, based on, for instance, descriptive tables; and specifying the relationships among the different elements for making possible their natural put into operation. In order to achieve the latter, the descriptors, not only common to all the theory elements, but also their own ones, should be identified if possible. The building of theories is a process and, in this case, the instrumentation is the next step.

In short, the building of theories requires patience, creativity, and persistence. And all of these requirements are worthwhile, as a theory creates a new way to look at the world [36], leads us to ask new questions, and compels us to embark on new and fruitful lines of inquiry.

Currently, theory in software development is as a child that has to grow up. It can be done by improving the holon/informon-based theory with practical usefulness and by putting it into operation in order to confirm/refute its predictions. Higgins [37] has used parenting as a metaphor for building and developing theories: “a theory, like a child, must be allowed to develop through contact with the world… To begin with, good parents do not assume what their child’s actual behaviours are like. They observe how their child’s actual behaviours in the world unfold in order to learn what their child is like. Similarly, scientists should not be [overly] concerned with confirming their theory’s predictions. They should observe the data produced by theory-driven research to learn more deeply about the theoretical mechanisms and processes”.

4 Conclusion

Hardware development has evolved exponentially largely thanks to the fact that it is supported by engineering based on science, e.g. physics, whose scientific theory is very well established. On the other hand, software development has evolved linearly, where the notorious lack of theory is possibly the primary reason for this trend.

Such thesis has been supported by the notable conceptual laxity existing, up to this moment, in software development with regards to the theory concept. That is, it seems that anything can be theory in software development. It doesn’t happen in the scientific fields, where there is a commonly accepted theory to explain and predict a type of phenomena. Then, if we want to help this discipline to advance in the scientific field it is necessary to complete it with the primary scientific basis, that is, the underlying theory.

This paper has established that a theory must explain the nature of phenomena (i.e., of information and its processes in software development) for, according that theory, perform effective predictions of these phenomena. Based on this definition, the holon/informon-based theory for software development is considered as a valid starting point. However, this proposal is stuck in abstract concepts without offering an operative specification for them. We propose that the instrumentation of that initial theory must be the next step for the researches.

In this way, it will be possible to advance towards the adoption of the scientific method in software development: observing the world, building its behavioral model based in a theory, assessing this model and, lastly, repeating the procedure after including the new knowledge.

References:


