e-Informatica Software Engineering Journal, Volume 6, Issue 1, 2012, pages: 71–77, DOI 10.5277/e-Inf120106

On Principles of Software Engineering – Role of the Inductive Inference

Ladislav Samuelis*
**Faculty of Electrical Engineering and Informatics, Technical University of Košice*
Ladislav.Samuelis@tuke.sk

Abstract

This paper highlights the role of the inductive inference principle in software engineering. It takes the challenge to settle differences and to confront the ideas behind the usual software engineering concepts. We focus on the inductive inference mechanism's role behind the automatic program construction activities and software evolution. We believe that the revision of rather ln old ideas in the new context of software engineering could enhance our endeavour and that is why deserves more attention.

1. Introduction - beyond software, beyond engineering

Before getting into details of the role of inductive inference in software engineering, we make some explanatory notes on the notions of *software* and *engineering* as it appears in the title of this contribution.

We can observe plenty of interpretations of the software and software engineering throughout the history of computing. Searching the origin of the word *software* F.R. Shapiro states that it appears for the first time in the work of John W. Tukey. He used that term in the context of computing in an article of the American Mathematical Monthly, in 1958! The quote is as follows [1, p. 1]:

> Today the 'software' comprising the carefully planned interpretive routines, compilers, and other aspects of automative programming are at least as important to the modern electronic calculator as its hardware of tubes, transistors, wires, tapes, and the like.

In 2008, the work of Leon J. Osterweil [2] appears. He suggests that there may exist other types of software besides computer software. He identifies parallels between computer software and other societal artefacts as laws, processes, recipes, instructions, and suggests that there are similar parallels in the ways, in which these artefacts are built and evolved.

There are 50 years between these two above-mentioned interpretations of the word *software*. The 'semantic gap' between the ideas that is behind this word has been widened enormously in time. The notion has gained more specific meanings during its life-span and it is expected to keep narrowing down. There are no signs that a rather 'calm' period of software 'evolution' would come. On the contrary, as we may conclude from the recent pervasively distributed and service-oriented software development.

Recently there has been an incredible increase in the performance of hardware. This increase itself is the reason for the incredible growth of software complexity. The Wirth's, or rather Reiser's 'law': *Software is getting slower, faster than hardware is getting faster* [3], allegorically points to the same fact.

After revealing partially the roots of the notion of software, we may focus on the notion of *engineering*. The question is as follows:

What feature of software enables us 'to engineer' it?

We can find a comprehensive answer to this question in the work of Wei-Lung Wang [4]:

The key reason is that software is a tangible form of mathematics that lends itself to being engineered. At its core, a program is an abstract sequence of instructions that performs some computation. But when the program is realized on a computer, it becomes an information tool with its own use-feedback cycle. It changes from an ethereal entity to a tangible tool and its actions can be observed. Instead of mathematically proving the results of a program, we can simply run it on some sets of inputs and observe its behaviour. This tangibility (or 'executability') is both software's strength and Achilles heel.

We accept that this specific feature of *observability in the material world* is the essence of the software *engineering*. In other words, this *tangibility* or *executability* enables specific ways of experimentation, which is the basis for observing the behaviour of the software by perception. In this way, we may create highly complex models that can be fully mapped into a computer representation and this model can be 'executed'.

Experiences gained from these observations also trigger needs for deeper understanding of the implemented ideas. We may say that observation by perception supports the comprehension of the modelled reality. This opposite process is *program comprehension*; when the task is to 'understand' (or to gain a mental image of) the computer model from the implemented code.

Of course, there also exist attempts that search a single unified theory of software engineering. For example, the contribution of P. Johnson and M. Ekstedt, presented at the recent International Conference on Software Engineering Advances [5], outlines the requirements for such a unified theory.

The statement of M. Jackson [6] is against a unified theory. He says that software engineering is a clumsy notion. He supports this observation with the fact that software engineering is split into various topics (e.g, compiler engineering, operating systems, database engineering, etc.) and in this way it does not cover any knowledge gained from solved problems. In other words, software engineering is an abstraction and every successful area of software engineering immediately changes to set up an independent specialization.

A recent article from M.S. Mahoney [7] characterizes the history of software engineering in the following way:

Historians and software engineers are both looking for a history of software engineering. For historians, it is a matter of finding a point of perspective from which to view an enterprise that is still in the process of defining itself. For software engineers, it is the question of finding a usable past, as they have sought to ground their vision of the enterprise on historical models taken from science, engineering, industry, and other professions.

These contradictory views and motivations of experts outline the broad diversity of research and ought to remind us to be careful with applying theory for building software.

2. The role of the inductive inference in automatic program construction

Automatic program construction has been a goal of the first programmer who faced with difficulties of programming. This activity has been spreading over the history of software engineering with various intensity and it acts like a moving target that constantly shifts in order to reflect changing requirements.

Much of what was originally conceived of as automatic programming was achieved a long time ago. In 1958 this term was mentioned for the first time in connection with the compiler construction. On the other hand, current expectations regarding its potential are often based on an idealized view of reality and some of them probably cannot be met. Nevertheless, a number of important developments are in progress in research efforts and in commercially available systems.

The term *automatic program construction* (or *program synthesis*) is used to refer to the study and implementation of methods for automating a significant part of the process of developing and maintaining software within the context of software life-cycle.

A broader goal of this field is to make computer programs significantly easier by means of automation selected software development process. More specific goals include increasing software productivity, lowering costs, increasing reliability, making more complex systems tractable, and allowing users to focus more on solving problems rather than on the details of implementation.

The big challenges for automatic program construction are defined, for example, by L. McLaughlin [8] in the following manner:

– to produce good runtime performance;
– to produce code that someone can look at, deal with, and understand;
– to ensure the code that is provably correct.

Every point deserves its own 'science' and the emergent research fields on automatic program synthesis follow roughly these criteria too.

The freedom of language selection allows using more declarative and less procedural specification. In other words, the specification is closer to the *what* (end-user defined) and the implementation is closer to the *how* end of the spectrum. A more technical characterization of the 'gap' between specification and implementation is that there is less detailed information content in the specification than in the implementation. The program synthesis process consists of filling in this gap with details that are omitted from the specification.

The automated synthesis of programs has its roots in artificial intelligence too. It is interesting to observe the mutual influence and synergy of ideas stemming from the field of software engineering and artificial intelligence. In particular, the task of inference of grammars from pattern analyses triggered the research on programming by examples [9].

In general, we distinguish two main methods in software development. *Deductive* methods - deduction is basically a problem of searching for an inference path from some initial set of facts to a goal fact. This fundamental mechanism is behind the deductive approach to automatic programming. In principle any method of automated deduction can be used to support automatic programming. For example, programming language PROLOG [10] (in fact, its inference engine) represents a deductive system. Despite its limitations, PROLOG remains among the most popular languages today, with many free and commercial implementations available. One of the challenges in research is to combine automated deduction with other methods.

Inductive methods - inductive inference is based in building models on the basis of experienced facts. Let us suppose that we have a mental model at our disposal. In the next step we may apply the standard scientific approach, which consists of *finding metrics*, *finding other descriptions* (or refinement of the model) and based on these new models *to infer the future behaviour* of the observed object. These three mental activities are, in fact, the inductive inference activities. In other words, finding metrics (or measurable features, data, observable signs, etc. on the model or on its output data) is indispensable for the description (or modelling). This activity is based on collecting individual data in order to create a hypothesis (model) with the process of generalization. We create model not only for the description of the actual systems but also for prediction. It means that the inferred model can serve for the prediction the system's behaviour in the future. An example of the application of the inductive inference is the 'programming by examples' where the examples serve for building models or rules (in this specific case a grammar).

Many areas exist where demonstration of an example is a suitable tool for automating tasks. For example, paths of robots represent linear plans and the task is to construct program or the sequence of learning objects represent the progress of the student in the learning material and the task is to construct the navigation plan (learning by watching or incremental learning). Programming by examples roots in the incremental learning, which was elaborated in 1970s and

1980s [11]. The structures of the systems devoted to synthesis of programs by examples are similar to the structure of linguistic pattern recognition systems [11].

It is evident that during the construction of the final model, a new instruction of the example could completely modify the existing model (due to the inductive inference principle). This fact deserves special awareness in the application of inductive inference.

In summary, knowledge of the part, which may be represented at whatever level of granularity, serve for the development of rules. In fact these rules may serve as basis for a new deductive system. For example, knowledge condensed in software design patterns represent higher level granularity of knowledge and these artefacts may also serve for building new system.

We may conclude that automatic program construction during the 1980s was more or less about the optimization of loops. In other words, the discovery and the implementation of reusable code segments was in fact the discovery of loops.

3. Notes on software evolution

In the 1950s, the term *automatic computing* referred to almost any work related with a computer. We tend to forget that before the object-oriented approach the methodology of automatic program construction was also associated with the idea of ‘construction of programs by examples’.

The research on automating programming before object-orientation was influenced to a great extent by results gained in artificial intelligence research ([12], and [13]).

There are plenty of articles and a special IEEE conference [14] devoted to software evolution. Research on software evolution is discussed in many software related disciplines. In any case, software evolution is equal to comprehension. This idea is briefly expressed by Jazayeri M. [15] as:

Not the software itself evolves, but our understanding and the comprehension of the reality.

This is in compliance with the idea that our understanding of the domain problem incrementally evolves and learning is an indispensable part of program comprehension.

The complexity of understanding and maintaining a program is proportional to its size and complexity. F. Brooks in the well-known paper ‘No Silver Bullet‘ [16], argues that programming is inherently complex. Whether we use a machine language or a high-level programming language, in general, we cannot simplify a program below a certain threshold that he calls an ‘essential program complexity’. The two main factors that determine this complexity threshold are:

– the complexity of a problem and its solution at the conceptual level, and
– the complexity of the infrastructure and the environment which has to be taken into account when solving problems by a program at operational level.

These two factors cannot be clearly separated from each other in program components, which constrains our efforts in using conventional decomposition (‘divide and conquer’) in combating software complexity. Very often, we cannot create a software conform like LEGO models unlike hardware constructions [17]. This difficulty of achieving clean decomposability is an important difference between hardware and software. Another important difference is that we do not try to change hardware so often and in such radical ways as we do software. The difficulty of clearly separating concerns by means of decomposability necessarily hinders changeability.

The inductive inference (or incrementality) appears in various contexts in the relatively short history of software engineering. This fact seems natural because software development processes like comprehension, design, refinement and realization are done iteratively and incrementally in practice. Due to this common fact incrementality notion is applied superficially in software engineering literature. We note that programming cannot be *fully* automated, since a computer must at least be told what to do. Only a human being is able to create ideas and tell it what to do. It is hoped that as technology progresses, the required details on how to do the task will steadily decrease.

4. The ubiquity of the inductive inference

The following remarks show the wide range of the usage of incrementality notion. A brief historical overview of the 'Incremental and Iterative Development' is presented in the work of C. Larman and V. Basili [18]. This work summarizes the role of the iterative and incremental software development through significant software projects since the mid of 1950s. It focuses on the incrementality utility, applied in software engineering processes, from the managerial point of view. It describes the driving thoughts and misbelieves, which were behind the practices applied in the past decades in the field of software engineering.

We accept in general that comprehension is also a continuous iterative and incremental process. The fact that problem solving does not progress in a linear manner from one activity to the next is highlighted as the conjecture:

Empirically based models mature from understanding to explaining and predicting capability.

This conjecture is explained in the handbook of authors A. Endres and D. Rombach [19, p.273], which is devoted to the empirical aspects of software engineering.

Inductive inference plays an important role in practical software engineering. At present time the incremental change in object-oriented programs are in focus (for example [20]). These activities investigate the impact of adding new functionalities into the code and finding the relevant program dependencies. Incrementality is important in software visualization too [21], where the aim is to get a better comprehension of the software behaviour by representing complex structures graphically.

The objective of the software development is to model a certain aspect or abstraction of reality as stated by B.Meyer in his work 'Reality: a cousin twice removed' [22]. Software engineering, as every engineering discipline, is characterized by trials and errors, which are necessary steps for clarifying the comprehension of the requirements, design and implementation. On the other hand we have to note that the incrementality principle has its mathematical roots and is explained in the theory of inductive inference [23]. Incremental software development is sometimes called 'build a little, test a little'. We can see the similarity between building concepts and models in software engineering and building hypotheses in mathematics. This process is very clearly highlighted in Pólya's classic work, 'How to Solve It' [24].

The empirical evidence from the real-world software suggests that learning or incremental program development is possible only when the data are presented incrementally. For instance, programming languages dispose with constructs, which help postpone solving some issues. A good example is the exception mechanism in object-oriented programming. This process makes, of course, the software more complex and drifts away from the original design. These facts may lower the quality of the software but it is the task of the validation and verification to ensure the formal quality software.

To sum up, the inductive inference is ubiquitous in software engineering. With each step we discover new requirements, analyze, plan, implement and test them. Every iteration adds new insights and the system grows or logically clarifies this way. In other words, software programs are too complex for getting correct details on any artefacts without some amount of experimentation. The software developers' ideas evolve as they progress.

5. Conclusion

This paper looks into the question: *What is the role of inductive inference in software engineering?* Its specific aim is to highlight the hidden role of inductive inference phenomenon behind the wide variety of software engineering concepts. It opens with automatic program construction and proceeds to the software evolution concept. Analysing the inductive inference in a sterile environment is not unusual. We take the challenge to settle out differences and confront the ideas behind the usual software engineering concepts in a turbulent and impure environment of recent software engineering activities. This approach might help in developing a more condensed foresight and provoke constructive thinking. We know that we did not invent a new solution to an existing problem; but we rather revised old ideas and analyzed them in recently applied software engineering practices. Practice generates always new problems and the task of software engineers is permanent strive for the identification essential concepts as it is expressed in the Semat initiative [25].

Acknowledgment

This work is supported by the Slovak Scientific Grant Agency: KEGA 040TUKE-4/2011: Modern software engineering in education - structure and implementation of software engineering subjects for university informatics study programmes.

References

[1] F. S. Preston, F. R. Shapiro, and L. R. Johnson, "Comments, queries, and debate," *IEEE Annals of the History of Computing*, Vol. 22, No. 2, pp. 69–71, 2000.

[2] L. J. Osterweil, "What is software?" *Automated Software Engineering*, Vol. 15, No. 3, pp. 261–273, 2008.

[3] N. Wirth, "A plea for lean software," *IEEE Computer*, Vol. 28, No. 2, pp. 64–68, 2006.

[4] W. Wei-Lung, "Beware the engineering metaphor," *Commun. ACM*, Vol. 45, No. 5, pp. 27–29, 2002.

[5] P. Johnson and M. Eckstedt, "In search of a unified theory of software engineering," in *International Conference on Software Engineering Advances, ICSEA*, 2007, p. 5.

[6] M. Jackson, "Will there ever be software engineering?" *IEEE Software*, Vol. 15, No. 1, pp. 36–39, 1998.

[7] M. S. Mahoney, "Finding a history for software engineering," *IEEE Annals of the History of Computing*, Vol. 26, No. 1, pp. 8–19, 2004.

[8] L. McLaughlin, "Automated programming: The next wave of developer power tools," *IEEE Software*, Vol. 23, No. 3, pp. 91–93, 2006.

[9] H. Liebermann, Ed., *Your Wish is My Command- Programming by Example, Automatic programming, Encyclopedia of Artificial Intelligence.* Morgan Kaufmann/San Francisco, February 2001.

[10] R. A. Kowalski, "The early years of logic programming," *Commun. ACM*, Vol. 31, No. 1, pp. 38–43, 1988.

[11] A. Cypher, D. C. Halbert, D. Kurlander, H. Lieberman, D. Maulsby, B. A. Myers, and A. Turransky, Eds., *Watch What I Do: Programming by Demonstration.* Cambridge University Press, 1993, ch. Programming by demonstration.

[12] J. S. Poulin, "Technical opinion: reuse: been there, done that," *Commun. ACM*, Vol. 42, No. 5, pp. 98–100, 1999.

[13] Z. Manna and R. J. Waldinger, "Toward automatic program synthesis," *Commun. ACM*, Vol. 14, No. 3, pp. 151–165, 1971.

[14] ACM, "IWPSE '01: Proceedings of the 4th international workshop on principles of software evolution," New York, NY, USA, 2001, conference Chair-Tamai, Tetsuo.

[15] M. Jazayeri, "Species evolve, individuals age," in *International Workshop on Principles of Software Evolution.* ACM, September 5-6, Lisbon 2005.

[16] J. F. P. Brooks, "No silver bullet essence and accidents of software engineering," *Computer*, Vol. 20, No. 4, pp. 10–19, April 1987.

[17] A. Ran, "Software isn't built from lego blocks," in *ACM Symposium on Software Reusability*, 1999, pp. 164–169.

[18] C. Larman and V. R. Basili, "Iterative and incremental development: A brief history," *Computer*, Vol. 36, No. 6, pp. 47–56, June 2003.

[19] A. Endres and D. Rombach, *A Handbook of Software and Systems Engineering; Empirical observations, laws and theories.* Pearson, Addison Wesley, 2003.

[20] V. Rajlich and P. Gosavi, "Incremental change in object-oriented programming," *IEEE Software*, Vol. 21, No. 4, pp. 62–69, Jul/Aug 2004.

[21] C. Knight and M. Munro, "Visual information-amplifying and foraging," in *Proceedings of SPIE, San Jose, USA, volume 4032. International Society for Optical Engineering*, January 2001, pp. 88–98.

[22] B. Meyer, "Reality: A cousin twice removed," *IEEE Computer*, Vol. 29, No. 7, pp. 96–97, July 1996.

[23] D. Angluin and C. H. Smith, "Inductive inference: Theory and methods," *Computing Surveys*,

Vol. 15, No. 3, pp. 283–269, September 1983.

[24] G. Pólya, *How to solve it: A New Aspect of Mathematical Method*, 2nd ed. Princeton University Press, 1957.

[25] I. Jacobson, "Discover the essence of software engineering," *CSI Communications*, pp. 12–14, July 2011.