Chapter XII
Notes on the Emerging Science of Software Evolution

Ladislav Samuelis
Technical University of Kosice, Slovakia

ABSTRACT

This chapter introduces the irreducibility principle within the context of computer science and software engineering disciplines. It argues that the evolution, analysis, and design of the application software, which represent higher level concepts, cannot be deduced from the underlying concepts, which are valid on a lower level of abstractions. We analyze two specific sweeping statements often observed in the software engineering community and highlight the presence of the reductionism approach being treated already in the philosophy. We draw an analogy between the irreducibility principle and this approach. Furthermore, we hope that deep understanding of the reductionism approach will assist in the correct application of software design principles.

INTRODUCTION

Dealing with continuously increasing software complexity raises huge maintenance costs and rapidly slows down implementation. One of the main reasons why software is becoming more and more complex is its flexibility, which is driven by changing business rules or other volatile requirements. We note that this flexibility is rooted in the generality of the programmable John von Neumann machine. Due to these inevitable facts, which influence software development, the software systems' complexity increases continuously. Recently, software maintenance represents 45% of software cost (Cartwright & Shepperd, 2000). This phenomenon motivates researchers and practitioners to find theories and practices in order to decrease the maintenance cost and keep the software development within reasonable managerial and financial constraints. The notion of *software evolution* (which is closely related and often interchanged with the term *maintenance*) was already introduced in the middle of the seventies when Lehman and Belady examined the growth and the evolution of a number of large software systems (Lehman & Belady, 1976). They proposed eight laws, which are often cited in software engineering literature and are considered as the first research results gained by observation during the evolution of large software systems.

The term software evolution has emerged in many research papers with roots both in computer science and software engineering disciplines (e.g., Bennett & Rajlich, 2000). Nowadays, it has become an accepted research area. In spite of the fact that the science of software evolution is in its infancy, formal theories are being developed and empirical observations are compared to the predicted results. Lehman's second law states the following: "*an evolving system increases its complexity unless work is done to reduce it*" (Lehman, 1980). Due to the consequences of this law and due to the increased computing power, the research in software and related areas is being accelerated and very often causes confusion and inconsistency in the used terminology.

This chapter aims to discuss the observations concerning evolution within the context of *computer science* and *software engineering*. In particular, it analyzes frictions in two sweeping statements, which we observe reading research papers in computer science, software engineering, and compares them with reality. We will analyze them from the reductionism point of view and argue that a design created at a higher level—its algorithm—is specific and in this sense it cannot be deduced from the laws, which are valid on more fundamental levels. We introduce a new term, the *irreducibility principle*, which is not mentioned explicitly in the expert literature within the context of *computer science* and *software engineering* (to the best of our knowledge). Finally, we summarize the ideas and possible implications from a wider perspective.

SOME HISTORICAL NOTES ON THE SOFTWARE EVOLUTION

Research on software evolution is discussed in many software related disciplines. Topics of software evolution are subjects of many conferences and workshops, too. In the following paragraphs, we will briefly characterize the scene in order to highlight the interpretation of the notion of evolution in the history of software technology.

The notions of *program synthesis* or *automated program construction* are the first forerunners of the evolution abstraction in software engineering. Papers devoted to these topics could be found in, for example, the research field of automated program synthesis. Practical results achieved in the field of programming by examples are summed up, for example, in the book edited by Lieberman (2001). The general principle of these approaches is based on the induction principle, which is analyzed in the work of Samuelis and Szabó in more details (Samuelis & Szabó, 2006). The term evolution was a synonym for *automation of the program construction* and for the discovery of *reusable code*—that is, searching for loops.

Later on, when programming technologies matured and program libraries and components were established into practice, the research field of *pattern reuse* (Fowler, 2000) and engineering *component-based systems* (Angster, 2004) drove its attention into theory and practice. In other words, slight shift to component-based aspect is observed in the course of the construction of programs. We may say that the widely used term of *customization* was stressed and this term also merged later with the notion of *evolution.* Of course, this shift was heavily supported by the object-oriented programming languages, which penetrated into the industrial practice during the 80s in the last century.

Since it was a necessity to maintain large and more complex legacy systems, the topic of *program comprehension* came into focus and became more and more important. Program comprehension is an activity drafted in the paper of Rajlich and Wilde as: Program comprehension is an essential part of software evolution and software maintenance: software that is not comprehended cannot be changed. The fields of software documentation, visualization, program design, and so forth, are driven by the need for program comprehension. Program comprehension also provides motivation for program analysis, refactoring, reengineering, and other processes. (Rajlich & Wilde, 2002). A very relevant observation concerning the comprehension is from Jazayeri who says "*Not the software itself evolves, but our understanding and the comprehension of the reality*" (Jazayeri, 2005). 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.

Rajlich, when dealing with the changing paradigm of software engineering, stresses the importance of the *concept location*. He argues that the volatility of the requirements is the result of the developer's learning. Thus, learning and understanding (or comprehension) are indispensably coupled with the evolution (Rajlich, 2006). We add that mental activities associated with understanding are dealt within the cognitive sciences and it is important to realize that, for example, software design concepts from certain higher level of abstractions cannot be formalized.

The scattered results from the mentioned areas lead to the attempt to establish the taxonomy of software evolution (Buckley, Mens, Zenger, Rashid, & Kniesel, 2005). Further areas of the contemporary research, which deal more or less with the evolution concept, are software merging (Mens, 2002), measurement of the flexibility and complexity of the software (Eden & Mens, 2006), and software visualization for reverse engineering (Koschke, 2000). It is also obvious that the evolution principle in the biological interpretation heavily attracted and influenced the research of the evolution in software engineering. The paper written by Nehaniv, Hewitt, Christianson, and Wernick critically warns the software engineering community about the non-obvious traps when the evolution principles valid in biology are mechanically applied to the area of software artifacts (Nehaniv et al., 2006). The analysis of these fields deserves special attention but examining them is not the aim of this chapter.

Summing up, the mentioned emerging disciplines are approaching the phenomenon of evolution from various aspects of the systems' analysis and design. These short introductory notes have glanced on the interlacing of software related disciplines and how they mutually influence each other. These approaches have their own history and theoretical roots; they are in various branches of computer science and treated from the philosophical point of view, too (King & Kimble, 2004). It is guaranteed that new techniques and research areas will emerge in the near future and further deal with the phenomenon of evolution.

SYSTEM, MODEL, AND REDUCTIONISM

Probably, the only unquestioned abstraction of the system theory is the *universe*, that is, the existing reality. This applies simultaneously and unambiguously to two things. It refers to the concept of the universe and the abstraction that our brain creates about it. We are able to deal with the things of the universe only through our abstractions about it and this is a dichotomy that we cannot solve with any argument.

Natural entities are existing units in reality, which we can select from the environment based on some *written* criteria. This unit may be for instance, an engine, a building, or a description of a complex banking system. A system is an entity when it comprises sub-entities.

The human cognition happens always through *system-models*. Modeling is always simplification and a kind of identification between two different systems. The cognitive human being creates a model in piecemeal growth. We use the model in a way that we *run* the model and this way we predict the modeled system's complex operation.

In the process of building the model, the following question rises naturally: Which features are stressed and measured in a model? In practice, this depends on the importance of a particular feature in a given context. In other words, we selectively omit features during the abstraction. We may also deliberately omit or intentionally extend specific features. This is part of the learning process when the acquisitions of new patterns are observed through experience.

New models can be created in a *revolutionary* or *evolutionary* manner. In essence, the difference between these two definitions lies in the fact that the non-incremental (revolutionary) approach is based on one-shot experience and the incremental (evolutionary) learning allows the learning process to take place over time in a continuous and progressive way, also taking into consideration the history of the training sets during building the inferred rules.

We may draw an analogy between these modes of models-creation and the definitions of the *s-type* and *e-type softwares* (Lehman & Ramil, 2001). Software evolution is only related to the *e-type* per definition. This type of software is influenced mainly with non-functional attributes as: reliability, availability, fault tolerance, testability, maintainability, performance, software safety, and software security. In other words, the e-type software has been influenced by unpredictable factors since the earliest stages of development and that is why it is continuously evolving without building it from the scratch.

We introduce the notion of *reductionism*, which frames the *irreducibility* principle. Reductionism generally says that the nature of complex things is reduced to the nature of sums of simpler or more fundamental things. In the following sections we will focus on and will clarify the two following sweeping statements that seem obvious at the first sight: (1) Theory valid within the computer science alleviates programming and (2) Programming is coding. They are often applied among the researchers and practitioners working in the field of computer science and software engineering.

STATEMENT 1: THEORY VALID WITHIN THE COMPUTER SCIENCE ALLEVIATES PROGRAMMING

It is often argued misleadingly in research papers that a specific theory valid within the computer science alleviates programming or software design. We introduce the notion of the *organizational level* or *levels of abstraction* and map the science valid within the computer to the lower level and the domain specific software to the higher level.

We argue that the sweeping statement: "*Theory valid within the computer science alleviates programming" represents* a typical reductionism approach. Actually, the analysis and design of software systems cannot be traced back, for example, to the state space of the computer's internal memory. The higher level of the abstraction has its own set of limits, which are domain specific and cannot be explained by the concepts, which are valid at lower levels. The limits of software design at higher levels cannot be reconstructed or predicted from the descriptions of abstractions, which are valid on lower levels. Naturally, we are able to follow causally the whole sequence of events, which happen on lower level, but this will not lead to solutions concerning design decisions

In other words, lower level laws equal the laws valid within a computer, and the aim of theoretical computer science is to reveal and establish the appropriate theories which describe the analyzed processes realized within the computer (e.g., automata theory). The functions valid at a higher level (the behavior and the algorithms) cannot be deduced back to a lower level. That is why it is an illusion that the paradigms valid within a computer can enhance the comprehension process and the programming of the system at the domain (higher) level. Of course, it is much better when somebody understands both levels of abstraction. The owner of such knowledge is in an advantageous position since the specific domain knowledge is definitely better underpinned.

Let us explain it through an analogy. As an example, we could mention the knowledge of the thermodynamics theory, which is valid within the cylinder of an internal-combustion engine, does not imply that we are able to construct an engine, which is produced for embedding it into a specific vehicle devoted to alleviate some work in a certain domain. It is also valid vice versa. It means that it is impossible to obtain the answer on the direction of vehicle's movement from the thermodynamic laws, which govern processes inside the cylinder. That is why we have to draw a sharp line between different abstraction (or organization) levels in order to feasibly argue and manage the relevant questions and tasks within that abstraction level (or domain). This is called the *irreducibility* concept, which is not a new idea in the theory of general evolution, as mentioned earlier.

In the context of software engineering, we can observe a similar situation; when applying the object-oriented approach to the analysis of a specific problem, we neglect the lower implementation level (the computer domain). Vice versa, when we investigate the domain of formal languages, then we neglect the application domain. Both considerations are also closely related to the pedagogical issue, which says that first we have to ponder about the domain problems and only later about the implementation details.

The mentioned sweeping statement can also be explained in philosophical terms, namely it is in accordance with the *reductionism* approach, when the nature of complex things is reduced to the nature of sums of simpler or more fundamental concepts. In the context of software, where we have several strata of the complexities (Mittermeir, 2001), it is obvious that the reductionism approach cannot be applied.

STATEMENT 2: PROGRAMMING IS CODING

This idea states that programming is a mere coding. It is far from reality that constructing programs equals coding. We are not going to summarize the historic milestones of software engineering but highlight that analyzing and designing software systems are essentially about building a model of reality, first of all. This activity is a complex set of actions and involves work of domain experts and specialists on the workflow management within the software project development.

Generally, software systems are determined by specifications or requirements, which are in fact sets of limits (barriers) on the required functionalities. We try to create a model of reality and to locate concepts. Yes, it is also valid in the opposite way. It is necessary to locate concepts in case we have a legacy code and would like to implement modifications. This includes activities related to program comprehension, which was mentioned earlier.

The core problem of the software development is getting an intellectual grasp on the complexity of the application. Software engineering is an empirical science because grasping requires experiments. Algorithms give answers and solutions for questions defined in a certain domain. These mechanisms are valid for a particular domain and they cannot be deduced from the underlying mechanisms valid within a computer. Mechanisms depend on a lot of project-specific criteria, such as the type, size, and criticality of application. It is not only the speed of running an application but, what is more important, the speed of developing reliable software functionality regardless of how fast it runs. From this point of view, almost any improvements in a piece of software could be viewed as an experimental debugging, which aims to improve the code.

We may turn back and stress that the experimental feature is inherent for the *e-type* software. The importance of the experiments or their ignorance, for example, Ariane flight failure (Lann, 1997) is underpinned with the already established experimental software engineering institutions throughout the world (Basili & Reiter, 1981). The more software is produced, the more its importance is increased on our everyday life. This fact highlights the dependency of the so-called information society on the reliable and robust software applications.

CONCLUSION

To conclude, knowledge of the laws, which govern on the level within the computer, cannot predict the aims (or outputs) of an application program, which is designed by the program (software) designer. The analysis and design of the system (on a higher organizational level) cannot be predicted from the behavior of the laws, which are valid on the lower organizational level. That is why mechanical transition of knowledge between the organizational levels is considered harmful and the knowledge transfer is *irreversible*. From another point of view, this is a kind of servicing, when a lower organizational level serves the higher organizational level. Services provided at the higher level cannot be deduced from mechanisms valid for lower levels (Bennett, 2004).

This chapter revealed rather old knowledge that was purified within the context of software related areas. We hope that the discussed ideas will focus the attention of software engineers towards reconsideration of the obviously claimed statements, which are against the facts observed in reality.

ACKNOWLEDGMENT

This work is supported by the Slovak Scientific Grant Agency: VEGA 1/0350/05—Knowledge-based software life cycle and architecture.

REFERENCES

Angster, E. (2004). SDP-city against a vicious circle! *First Monday, 9*(12). Retrieved February 28, 2004, from http://firstmonday.org

Basili, V., & Reiter, R., Jr. (1981). A controlled experiment quantitatively comparing software development approaches. *IEEE Transactions on Software Engineering, 7*(3), 299-320.

Bennett, K. (2004). Software evolution and the future for flexible software. Paper presented at the Erasmus University of Bari.

Bennett, K., & Rajlich, V. (2000). Software maintenance and evolution: A roadmap. In A. Finkelstein (Ed.), *The future of software engineering* (pp. 73-90). Limerick, Ireland: ACM.

Buckley, J., Mens, T., Zenger, M., Rashid, A., & Kniesel, G. (2005). Towards a taxonomy of software change. *Journal on Software Maintenance and Evolution: Research and Practice, 17*(5), 309-332.

Cartwright, M., & Shepperd, M. (2000). An empirical investigation of an object-oriented software system. *IEEE Transactions on Software Engineering, 26*(8), 786-796.

Eden, A. H., & Mens, T. (2006). Measuring software flexibility. *IEE Software, 153*(3), 113-126.

Fowler, M. (2000). *Analysis patterns: Reusable object models.* The Addison-Wesley Object Technology Series.

Jazayeri, M. (2005, September 5-6). *Species evolve, individuals age.* Paper presented at the International Workshop on Principles of Software Evolution, ACM, Lisbon.

King, D., & Kimble, C. (2004). Uncovering the epistemological and ontological assumptions of software designers. In *Proceedings 9e Colloque de I'AIM*, Evry, France.

Koschke, R. (2002). Software visualization for reverse engineering. In S. Diehl (Ed.), *Springer lecture notes on computer science (LNCS) 2269: Software visualization, state-of-the-art survey.*

Lehman, M. M. (1980). On understanding laws, evolution, and conservation in the large-program life cycle. *Journal of Systems and Software, 1*(3), 213-231.

Lehman, M. M., & Belady, L. A. (1976). A model of large program development. *IBM Systems Journal, 15*(3), 225-252.

Lehman, M. M., & Ramil, J. F. (2001). *Evolution in software and related areas.* Paper presented at the workshop *IWPSE* (pp. 1-16), Vienna Austria.

Le Lann, G. (1997). An analysis of the Ariane 5 flight 501 failure—a system engineering perspective. Paper presented at the 10th IEEE Intl. ECBS Workshop (pp. 339–346).

Lieberman, H. (Ed.). (2001). *Your wish is my command, programming by example*. Media Lab., MIT, Academic Press.

Mens, T. (2002). A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering, 28*(5), 449-462.

Mittermeir, R. L. (2001). Software evolution—let's sharpen the terminology before sharpening (out-of-scope) tools. Paper presented at the Workshop *IWPSE* (pp. 114-120), Vienna, Austria.

Nehaniv, L. C., Hewitt, J., Christianson, B., & Wernick, P. (2006). *What software evolution and biological evolution don't have in common?* Paper presented at the Second International IEEE Workshop on Software Evolvability, IEEE, Philadelphia, Pennsylvania, USA.

Rajlich, V. (2006). Changing the paradigm of software engineering. *Communications of the ACM, 49*(8), 67-70.

Rajlich, V., & Wilde, N. (2002). The role of concepts in program comprehension. In *Proceedings of the IEEE International Workshop on Program Comprehension* (pp. 271-278). IEEE Computer Society Press.

Samuelis, L., & Szabó, C. (2006). Notes on the role of the incrementality in software engineering. *Univ. Babes, Bolyai, Informatica*, *LI*(2), 11-18.

KEY TERMS

Automatic Programming: The term identifies a type of computer programming in which some mechanism generates a computer program rather than have human programmers write the code.

Induction: Induction or inductive reasoning is the process of reasoning in which the premises of an argument are believed to support the conclusion but do not ensure it. It is used to formulate laws based on limited observations of recurring phenomenal patterns.

Irreversibility: Processes in general that are not reversible are termed irreversible.

Reductionism: Is a philosophical theory that asserts that the nature of complex things is reduced to the nature of sums of simpler or more fundamental things.

Reusability: Reusability is the likelihood a segment of source code can be used again to add new functionalities with slight or no modification. Reusable modules and classes reduce implementation time, increase the likelihood that prior testing and use has eliminated bugs, and localizes code modifications when a change in implementation is required.

Software evolution: Software evolution refers to the process of developing software initially, then repeatedly updating it for various reasons.

Synthesis of Programs: Comprises a range of technologies for the automatic generation of executable computer programs from high-level specifications of their behavior.