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 com-