Crisis Response and Management

Sergey V. Zykov
National Research University Higher School of Economics, Russia

INTRODUCTION

In the 1960s, the so-called "software crisis" triggered the advent of software engineering as a discipline. This term originated from the critical development complexity, which happened due to the rapid growth of computational power. At that time, the computing power of the machines became so overwhelming that a number of software development projects were over budget, late or unsuccessful. One well-known example was the first General Electric's payroll system launched in 1954 at Louisville, Kentucky; this was late, over budget, and missing crucial features (Topi, & Tucker, 2014). Irrespective of human efforts, the complexity of the hardware and software systems was hard to cope with by means of the old methods and techniques. The challenge was so dramatic that in 1967 NATO arranged an invitation-only conference, where world leaders in IT research and practice searched for an efficient response. At the conference, the term "software crisis" was coined by F.Bauer and used by E.Dijkstra (Naur, & Randell, 1968).

Another term suggested at the conference by the same F. Bauer was software engineering. The idea was to apply the engineering methods of material production to the new domain of large-scale concurrent software systems in order to make the software projects more accurate and predictable. This software engineering approach was feasible, though the methods and practices used had to differ substantially from those used in the material production. Specifically, the experts examined bridges as the instances of complex material systems.

The attendees concluded that the distribution of time and cost by the lifecycle phases, especially for the post-delivery maintenance was very different for software and material production. This is why the new software engineering discipline was in need of new methodologies, techniques and tools.

The focus of the software engineering discipline was the "serial" production of substantially large-scale, complex and high quality software systems. Concerning software complexity, at least two dimensions were identified; these were technical and management complexity (Booch, 2006). To measure software product complexity and quality, a set of attributes and metrics was suggested. The quality attributes included performance, reliability, security, fault tolerance, usability, strategic reusability and maintainability; their importance depended on the product size and scope (Lattanze, 2008). The complexity metrics included product size in terms of lines of code, function points, nesting levels, cyclomatic complexity and a number of more sophisticated ones (Debbarma, Debbarma, Chakma, & Jamatia, 2013). These metrics assisted in the divide-and-conquer strategy; later, they this general approach transformed into elaborate product estimation techniques and software development methodologies (Jensen, 2014).

Researchers argue whether the crisis in software engineering is over yet (Colburn, Hsieh, Kehrt, & Kimball, 2008) or it still exists (Buettner, Dai, Pongnumkul, & Prasad, 2015). This happens because of the fundamental differences in the lifecycles of software and material products. One critical difference between large-scale software and material production is the distribution of time and cost by the development lifecycle phases. Therewith, maintenance is the most time and cost consuming, it often exceeds 60% of the software project expenses (Schach, 2011). The other crucial difference is that

DOI: 10.4018/978-1-5225-2255-3.ch120

software production often depends dramatically upon human factors. These human factors relate to the management aspects of software complexity, whereas the technology factors relate to the technological aspects. Certain product categories are far more complex in terms of management than in terms of technology; however, the influence of the human factors on their development is largely underestimated. For such software product categories as enterprise information systems and defense management information systems, neglecting these human factors often results in project delays or even failures (Booch, 2006).

Therewith, the software crisis originates from a number of factors; these are human-related and technology-related factors. To manage this crisis, the authors suggest a set of software engineering methods, which systematically optimize the lifecycles for both types of these influencing factors. This lifecycle optimization strategy includes crisis-responsive methodologies, system-level architectural patterns, informing process frameworks, and a set of knowledge transfer principles (Zykov, 2009; Zykov, Shapkin, Kazantsev, & Roslovtsev, 2015; Zykov, 2015).

Software development usually involves customers, developers and their management; each of these parties has different preferences and expectations. Therewith, these parties often differ in their vision of the resulting product; typically, the customers focus on business value while the developers are concerned with technological aspects. Such a difference in focus often results in crises. Thus, the software crises often has a human factor-related root cause. To deal with these kind of crises, software engineers should enhance their skillset with managerial skills, such as teamwork, communications, negotiations, and risk management.

BACKGROUND

In the 1960s, the software production lifecycle was unstable as no systematic approach existed. At that time, software development did not allow for adequate planning and management of a project timeline and budget. The software products were unique masterpieces; they used a build-and-fix approach as the core "methodology". A systematic approach to product lifecycle was required in order to manage this crisis of development. This approach was to include certain technical and management aspects. The technical aspects should include justified architecture selection and high-level design. The management aspects should include teamwork and transparent communication between the client and the developer.

In this period, software development involved a number of parties with significant differences in goals and expectations. These were clients, developers and their management. Each side usually had a different understanding of the future product, as the clients were business oriented, while the developers focused on technology.

At present, this same lack of common understanding hampers software development; it is a possible source of a software crisis resulting from management complexity. To deal with this kind of crisis, software engineers enhance their technical abilities by a specific "soft" skillset. This includes collaborative teamwork, risk management, communications and negotiations. The "soft" skills assist software engineers in their management of the present-day software development crisis, which often results from human factors.

The following decades of the 1970s and 1980s changed software development from an art to a science, though it had not yet become a serial production technology. Techniques and methods appeared which are currently known as the Programmer's Workbench (Dolotta, & Mashey, 1976), third-generation programming languages and the supporting Structured Analysis and Design Technique (Dahl, Dijkstra, & Hoare, 1972). Further, in the 1990s software development technologies became even more advanced. The new technologies which arose at that time were more process focused; a few examples of these include OOAD (Jacobson, Christerson,