We are going to focus on the reasons for the crisis in software development, which occurred relatively long ago, in the 1960s. We analyze the reasons for the crisis. Software product lifecycles, which the industry had just started moving toward, were anarchic in many ways, since no systematic approach existed.

At that time, software development did not allow precise variation of such basic project parameters as time, cost or functionality. In fact, the software products were unique masterpieces, with a build-and-fix approach as the core “methodology”. Thus, a systematic approach to product lifecycle, and responsibility for the deliverables was required. During the following decade, the software development process gradually became a science rather than an art; however, because of imperfect technologies it had not become a production yet. Large-scale software re-search and development centers appeared, such as the Software Engineering In-stitute of Carnegie Mellon University. The value of software increased compared to hardware. Mission-critical software systems appeared, such as military and life-support applications. However, the software crisis lasted much longer and had a deeper nature than that in material manufacturing industries. Currently, the lack of a universal methodology for software development explicitly indicates that the crisis is not completely over. To manage the crisis, we need to optimize the soft-ware product lifecycle. Software engineering methods can help in dealing with the crisis, since this discipline systematically approaches software development issues.

The software engineering approach is chiefly oriented on “serial production” of large-scale, complex, high quality, architecturally heterogeneous and interopera-ble software systems. Other architectural aspects include portal-based software systems, remote services, and the like. Software product quality is measurable by a number of “dimensions” or attributes, such as performance, reliability, security, fault tolerance, usability and maintainability. Heterogeneous software systems incorporate versatile architectures, databases and data warehouses, which include both strong- and weak-structured data.

The global economic crises and the subsequent depressions taught us certain les-sons. We present an adaptive methodology of software system development which allows avoiding local crises, specifically large-scale ones. The methodology is based on extracting high-level common enterprise software patterns and apply-ing them to a series of heterogeneous implementations. The approach includes a new model, which extends the conventional spiral lifecycle by formal models for data representation and management, and by domain-specific language-based CASE tools. The methodology application areas include oil-and-gas resource planning, air traffic control, and nuclear power production. Further, we discuss possible combinations of software lifecycle models and examine the factors for crisis-based terms and cost reduction. Another area of interest is the adjustment of the software lifecycle according to project size and scope. Therewith, the so-called “human factor errors” resulting from crisis conditions and non-systematic software lifecycles, and their impact on a crisis, are analyzed. The book outlines the ways to systematic and efficient software-related project lifecycles, and sug-gests certain troubleshooting methods. Let us focus on the reasons for the crisis in software development.

The crisis in software development started relatively long ago, approximately in the 1960s. Let us analyze the reasons for the crisis.

At that time, software product lifecycle, which the industry had just started moving toward, was anarchic in many ways, since there was no uniform, systematic approach to software development in terms of product and process. Software development did not allow precise monitoring and adjustment of the major project parameters, such as time, budget and product quality. In fact, it was hard to manage the software development, as the build-and-fix approach was the key methodology. Because of the absence of proper standards, metrics and rocketing complexity of the computer hardware and the related software development tasks, the software development lifecycle was hard to manage, predict and control. The following decade of the 1970s showed that the software development process was gradually becoming a science rather than an art; however, because of imperfect technologies it did not become a production process yet. The era of hand-made software products from unique gifted programmers was over.
A few large software research and development centers appeared; one of the most well-known examples is the Software Engineering Institute of the Carnegie Mellon University [24].

The value of software, as compared to hardware, increased tremendously. Mission-critical software systems appeared to monitor and control military and life-support applications.

However, the software crisis, which started in the 1960s, lasted much longer and probably had a deeper nature than that in material manufacturing industries, such as construction, automobile production and the like. The absence of a relatively universal methodology, the so-called “silver bullet” for software development, explicitly indicated that the crisis was still there.

To manage the crisis, we need to optimize the software development lifecycle by systematically approaching all of its processes and the resulting product quality attributes. Since software engineering addresses the software development issues in a systematic way, we assume that its methods and tools are probably useful for the crisis management.

To understand the nature of crisis, we will analyze a number of patterns for software development lifecycle, each of which has certain phases or stages. Of these, we will identify the most mission-critical phases in terms of resource consumption and management dependencies, and see in what way they are different from the material production. We will see that irrespective of the product size and scope, software production has a number of fundamental differences as compared to material production. For example, software development lifecycle is often essentially shorter than that of a material object, since software products usually morally degrade much faster than the material ones. Another consideration is that the software product changes are often more serious and radical than the changes in material objects. For instance, there is a number of buildings and bridges used for many decades at relatively small or even negligible maintenance costs as compared to software.

Brute force approach is not quite applicable to software. For example, doubling a data channel throughput would not guarantee its reliability. However, making a bridge trestle two times thicker would result in a deliberately reliable product.

In contrast to material production, a software product model and its prototype should not necessarily be reliable. Any operational software release usually contains a certain number of defects; however, it is not catastrophic for the product, as it is in the case of a certain automobile model. Even after a serious software crash, it is often sufficient to just restart the product rather than to repair or reproduce it. Software defects tend to accumulate in time. Defect detection is a challenge, and building defect-free software systems requires different methods as compared to material objects construction.

In addition to the lifecycle patterns, or models, we will consider the software development methodologies, which include processes, roles, methods and tools. According to each process step, which consists of larger workflows and smaller activities, every role is to produce a certain artifact or deliverable that is a part of the software product. These deliverables include not only software code, but also documentation artifacts, which are equally important for the product quality.

Software product development usually requires a team, especially in case of large-scale and complex products. However, the accumulated team experience in software development does not always result in system quality increase. This often happens because of rapid changes of complex software platforms; another important reason may be the human-related factors, which depend on the so-called “soft” skills, such as communication, negotiation, self-reflection and self-adjustment.

The software development lifecycles are often iterative, incremental and reuse-oriented. However, we will also consider straightforward one-pass and abbreviated processes, which proved crisis efficiency under certain conditions.

Concerning the software engineering methodologies, we are going to analyze a number of process frameworks for software development. Though all of them are potentially adjustable, we will identify and compare formal and agile methodologies in terms of crisis applicability for general-purpose and mission-critical software product development.

We will then discuss patterns and practices of the methodologies application to real-world implementations. These embrace several industries including civil air transportation, oil-and-gas production and nuclear power plant construction, to name a few.
To address the human-related factors, which may distort common vision of the product by the client’s and the developer’s sides and thus initiate a crisis, we propose a number of models and techniques. These include informing science approach for efficient communication modeling, and a set of psychologically approved practices for knowledge transfer management.

Software development and material production have a number of similarities and essential differences. Thus, we need to use software engineering models, methods and tools to provide predictable and measurable product development in terms of quality. This is mission-critical for industrial production of large and complex software systems with such quality attributes as availability, maintainability, reliability, security and reusability. In crisis, however, software engineering methods and tools provide a rigorous technology basis required even for general-purpose software production.