Preface

“Any intelligent fool can make things bigger and more complex... It takes a touch of genius - and a lot of courage to move in the opposite direction.” – Albert Einstein

The ever-expanding field of Software Engineering continues its journey with the introduction of emerging topics and concepts in order to solve particular problems of the modern software industry. As one of the major fields, it has grown rapidly in recent years, in parallel with the complexity and scale of the software systems. Addressing a wide spectrum, from traditional engineering systems to the large, Web-scale architectures of today, customer requirements have always given rise to innovative approaches to be developed either through introducing new paradigms or extending the capabilities of well-established approaches. Therefore, a rapid explosion of advanced techniques, methodologies and tools has been witnessed in the recent years, especially when the well-established principles of the past are applied to today’s upcoming challenges in order to build more reliable, evolving, and professional software systems.

A modern perspective to software engineering requires an outlook to understand those concepts and principles that have matured over the years as the field evolves. Actually, those principles are transferred from older terminology to newer ones: concepts such as abstraction, modularization, separation of concerns, composition, process improvement, adaptation, testing and so on have always been part of the life of a software engineer. What else shape and distinguish a modern perspective, in fact, are the key challenges facing software engineering. Being one of the major fields, software engineering research has expanded rapidly in recent years, in parallel with the complexity and scale of the software systems. As of today, software becomes an integral part of various systems including automobiles, aircrafts, World-Wide-Web, or even tiny devices. Therefore, many frontier approaches have either emerged as a remedy for the engineering needs of the software community or gained more momentum than they have in the past. The modern software engineer cannot afford to quest for incremental gains only—substantially different instruments are needed for a leap in the performance of the industry.

Whenever a new approach emerges, the software industry is also thirsty for an effective, yet simple and published method to safely utilize the approach. For this purpose, enactment of an emerging approach in the solution of a real world problem is as significant as the approach itself. The struggle is not merely for the sake of being up to date or feeling so; the software engineering world is really in need of approaches that are practically available to a broad range of practitioners. Real world practices, case studies, tools and development environments are all critical for the adoption of new techniques, in an effort to increase the quality of software.

In this book, we aim to provide an outlook to these recent concepts and approaches. There are two unique features of the book which deserve particular emphasis. The first is its broad coverage of the topics...
that provides a gateway to current directions in modern software engineering from both a researcher’s and a practitioner’s point of view. Even though it is impossible to cover all topics in the field, it combines various techniques into one coherent structure and presents them in a concise manner. Furthermore, the chapters devoted to the topics not only present the concepts, but also provide a clear outlook over their position in the field with its newly evolving personality.

Another important feature of the book is its new orientation that accommodates modern techniques. For this purpose, a range of new concepts occupying the literature are incorporated within a composition, enabling them to be accessible to engineers. Being more important than mere utilization of modern techniques, the long mentioned need in the software engineering world for this kind of composition is finally made practically available. Many authorities have been reporting the need for this kind of methodology, and many have proposed bits and pieces of this idea with different terminology. This line of thinking is also to show the expansion in the field and also indicates that they are not isolated islands but connected to a large body of land. We believe that the contribution of the book is unique in terms of compiling such approaches that will eventually find their way to the industry and change the course of software development drastically.

A MODERN PERSPECTIVE

A natural question arises: what does a modern perspective to software engineering constitute? A straightforward answer to this question could be to enumerate all recent topics that we deal in software engineering today. Such an answer might, however, be misleading without knowing how and why they have evolved. In the following, we discuss what the main drivers are, why they are critical to be a major motivation in a modern perspective, and how the emerging approaches contribute to the challenges we have today.

The Big Challenge

Despite being an essential asset in everyday life, software systems are still facing critical problems in terms of increasing size and complexity, overdue and over budgeted projects, or unreliable operation performance. According to statistics over software project failures, the Chaos Report by the Standish Group\(^1\) states that, only in 1995, U.S. companies and government agencies spent $81 billion for canceled software projects and an additional $59 billion for software projects that were overdue. These figures have not improved so much over the last decade: According to the Chaos 2009 survey\(^2\), there is a noticeable decrease in project success rates, with only 32% of all projects succeeding, and 68% of those are either challenged (late, over budget, or with less than the required features and functions) or failed. The grounds yielding such an undermined picture could be many. One can directly relate it to inherent complexity of software systems to be the main reason, considering that today’s systems need to operate in ever more pervasive environments in a highly distributed or embedded fashion. Software in such environments is often required to ensure reliability, security, and performance. There is, as well, an elevated demand for the software to be evolving and adaptive, complying with the changes in business, technology, or legal issues. Despite significant advances in programming languages and supporting integrated development environments (IDEs), the problem is more than merely code-centric, but involves architectural, operational, managerial, and organizational aspects. Nonetheless, most of these challenges
are not quite addressable during the lifetime of an individual software project. Rather, well-established methodologies, mature tools, industrial best practices, and more importantly, theoretical research are required to pinpoint the problems and to offer innovative solutions.

**Strive for Simplicity**

The growing complexity of software systems is a key motivation behind the recent techniques to simplify design and development. On one hand, Model-Driven Engineering (MDE) evolved as an idea of creating models, or abstractions, concerning particular concepts and roles rather than directly modeling the code. This is intuitive when one considers the whole software design process as a concern-oriented, multi-leveled process where, in each level, engineers are supposed to transform problem-specific concerns into solution-specific ones. This simplifies the process of design as it provides a systematic transformation of problem-level abstractions to software implementations. Each level can also be supported with automation via tools, transformation and generators decreasing the complexity of bridging the gap between the levels. In such a vision of software development, models are the primary artifacts of development and lead the way also to higher level abstractions for domain-specific modeling.

On the other hand, lightweight methodologies (i.e. agile development) have gained more popularity in the last decade as a remedy to increased bureaucracy and slow pace of preceding methodologies. They work quite well, especially, when software project is not mission-critical, requirements change rapidly, and the development team is relatively small. The success of agile development has triggered some attempts in the software community to derive software development methods (e.g. Agile Unified Process) by incorporating the agile principles to well-known, plan-driven methodologies in order to create a synergy of both approaches. In addition, the agile principles are closely related to Lean Manufacturing and Six Sigma which are two widespread practices in operations management. Such an interdisciplinary approach can provide additional benefits for software development process as those principles of Lean Manufacturing and Six Sigma have already shown their applicability by helping to revolutionize the manufacturing process.

**Predictable Economics**

Economics is a significant determinant factor for the success of any software project and its value for a company. In order to maximize the Return of Investment (ROI) of a project, software engineers need to carefully consider and plan the cost of production. If not planned and foreseen beforehand, the cost can change exponentially and result in software project failures whose results might be devastating. In this regard, software economics appears as a field that seeks enabling improvements to software design and development via better predicting the size, complexity and cost of a software process, and enabling economic reasoning about the final product. Although it is a relatively mature sub-field with a significant amount of effort already spent, increasing demand for better cost predictions, and accurate scheduling still makes software economics a very active area.

A variety of advances took place that enable economic reasoning in a software project. Although these advances differ significantly in terms of the measures and parts that they improve in software project, their contribution to the decision-making in software economics is almost of equal importance. First, the size and complexity of software to be developed is a critical factor for ensuring an effective decision making and the focus of Software Cost Estimation (SCE), which predicts the effort and time
required for a software project. SCE involves the construction of an estimation model which is mostly a statistical technique for a theoretical measurement and interpretation. These estimations are crucial for early design decisions and cost predictions, and, in turn, an essential input into any cost-benefit analysis.

Whereas size of software is crucial from an economical point of view, another point of research deals with Software Process Improvement (SPI) that explores the ways of improving the development process. In the last decade, we observed advances towards more managed processes and their widespread adoption in the organizations. The process and maturity models (e.g. CMMI, SPICE) developed for this purpose help the organizations to minimize risks and increase competitiveness. One challenge is to incorporate more agility and adaptability in a managed process constituting a crucial target for the research community. Another important focus is the development of effective design tools that increase the productivity and automation in a software process. Improvements in these directions have direct impact on making the software economics more predictable and to maximize the ROI of software projects.

**Pushing the Limits**

Recent advances in network technologies and the increasing popularity of the World Wide Web define new frontiers for software engineering. As the Web revolutionizes business and lifestyle with drastic changes, software engineering for Web-based applications is one of the limits that we have challenged in the recent years. From a design perspective, developing Web based applications involves significantly different practices when compared to developing traditional applications. Multi-users, scalability, flexibility, security, and trust are only some of the requirements that gain increasing importance when applications are developed and deployed for the Web. In addition, recent notions Software-as-a-Service and Cloud Computing offer a new perspective to consider the software not as a product, but as a service, to be deployed and used as a utility. Design and development of software for this purpose generates a number of difficulties that already find their way in challenging the software developers.

Another shift is taking place in the computer hardware field while the era of doubling performance periodically in the chip industry is about to come to an end. Multicore chips are emerging while pushing parallel computing into the mainstream. Due to the affordable prices and future predictions of chips with many cores, the embedded software to operate on these architectures is gaining significance in software industry. Unfortunately, one cannot rely solely on low level compilers to perform the parallelization; it is the parallelization of software that is noteworthy when performance is considered. Therefore, the challenge for software engineering becomes how to develop software to operate better on multicore architecture while minimizing design complexity resulting from parallelism.

**BOOK ORGANIZATION**

This book addresses many of the abovementioned topics in a modern perspective to software engineering and introduces and discusses concepts and practices for successful software development. In the remaining part of this preface, we briefly introduce summaries of the sixteen chapters of the book.
Chapter 1. A Comparative Analysis of Software Engineering with Mature Engineering Disciplines Using a Problem-Solving Perspective

Software engineering is in essence a problem-solving process, and to understand software engineering it is necessary to understand problem-solving. To grasp the essence of problem-solving, we have provided an in-depth analysis of the history of problem-solving in mature engineering and software engineering. This has enabled us to position the software engineering discipline and validate its maturity level. To explicitly reason about the various problem-solving concepts in engineering, we also introduce the Problem-solving for Engineering Model (PSEM) that uniquely integrates the concepts of problem-solving, control and context. It appears that mature engineering conforms to the PSEM and this maturation process has been justified by a conceptual analysis from a historical perspective.

Chapter 2. Is Lean Agile and Agile Lean? A Comparison between Two Software Development Paradigms

This chapter provides a comparative study of lean and agile development methodologies that are emerging as answers to agility needs of organizations. The study shows that the principles of these approaches agree on the goals to be achieved, while lean development stresses an end-to-end perspective. The chapter also indicates research implications to enable a synergy of both approaches and to maximize the benefits of industry by their adoption.

Chapter 3. OntoArch Reliability-Aware Software Architecture Design and Experience

Ensuring software reliability prior to the development draws increasing attention, as the design decisions have direct impact on software system’s capability to meet reliability requirements. This chapter presents the OntoArch method in order to create a reliability-aware architecture design. OntoArch incorporates quantitative reliability evaluation in software architecture design by means of an ontology-based approach and the underlying tool. It incorporates quantitative software reliability measurement into software architecture design and enables reliability experts to formally conduct architecture modeling and reliability modeling in a unified environment. The approach is validated for a software architecture design, i.e. Personal Information Repository (PIR), with the use cases of software architecture knowledge management, software reliability profiling, and software architecture modeling and evaluation.

Chapter 4. Architecture-Driven Modernization

One of the critical features of a software system is to support its evolution over time in order to adapt the system to the changes in technology and business. When such a feature is missing or not well-designed, a software system becomes prone to be outdated. Today, such systems, namely legacy systems, pose a critical problem because it is usually not so trivial for the enterprises to replace them due to economical and technical reasons. Software modernization mainly seeks the means to migrate a legacy system into a new form by examination, and modifications of system aspects. This chapter presents a recently adopted approach for software modernization, namely Architecture-Driven Modernization, which advocates carrying out reengineering processes taking into account the principles and standards of model-driven
development. An overview of Architecture Driven Modernization shows how it allows legacy information systems to evolve while making them more agile, preserving the embedded business knowledge, and reducing their maintenance costs.

**Chapter 5. Architecture-Centered Integrated Verification**

Software architecture specifies a high-level abstraction of system topology, functionality, and behavior as a basis for system understanding and analysis. This chapter presents an architecture-centered verification approach to large scale complex software systems by integrating model checking with runtime verification. The approach aims at providing analysis and verification in two different levels of software development process—design level and implementation level—hence, bridging the gap between software architecture analysis and verification and the software product.

**Chapter 6. Modeling Services Using ISE Framework: Foundations and Extensions**

The Internet revolutionizes deployment and delivery of software to a great extent. In this regard, the Internet of Services (IoS) is a new paradigm that introduces new requirements for service engineering in terms of addressing both business and technical perspectives. This chapter introduces a novel service engineering framework, Integrated Service Engineering (ISE) framework, and its ISE workbench in order to address these emerging requirements of IoS. The chapter provides a comprehensive overview of the foundations of IoS, and the principles on how the service engineering process can be conducted by applying separation of concerns. Additionally, three novel extensions are presented to the underlying ISE workbench in order to enrich the capabilities of the service modeling process.


Semantic service descriptions are shown to be useful to describe several service properties with rich semantics and expressive syntax. The expressiveness of these descriptions enables the automation of many service-related tasks (e.g., discovery, selection, composition), and leads the way to adopt many innovative techniques for a seamless interaction in a large service ecosystem. However, when a service is composed from several other services in a dynamic fashion of a service ecosystem, the description of the service becomes a challenging task as many non-functional service aspects such as guarantees, pricing, payment, penalties, or delivery modes are hard to determine. This chapter provides an approach called Visual Semantic Analysis (VSA) in order to support semi-automatic modeling of semantic service descriptions for addressing such issues.

**Chapter 8. Description, Classification and Discovery Approaches for Software Components: A Comparative Study**

Software components constitute a well-established technique to facilitate reuse and composition. Many techniques are proposed up to today that differ in terms of description, classification, and discovery of components. This chapter presents a comparative study of component-based approaches and identifies
the key factors for a successful repository reuse. The comparison is based on search style, information type, comparison distance, and specification level, and highlights the importance of non-functional constraints in component descriptions, advantage of approximate comparison, and also trade-off between the level of specification detail and the degree of difficulty to discover a component.


Cost estimation is a challenging task for software engineers to estimate the time and cost of a software project, especially when the complexity and the size of the software is large. Recently, the use of historical data, along with statistical estimation techniques, has gained special attention. A common problem in building software cost models is that the available datasets may contain projects with lots of missing categorical data. This chapter presents a comprehensive framework showing that a combination of modern statistical and computational techniques can be used to investigate the effect of missing data on the accuracy of cost estimation. In addition, a recently proposed missing data technique, the multinomial logistic regression, is evaluated and compared with existing methods such as listwise deletion, mean imputation, expectation maximization, and regression imputation with respect to their effect on the prediction accuracy of a least squares regression cost model.

Chapter 10. Formalization Studies in Functional Size Measurement

Software metrics are the key to quantitatively measure and estimate the software or its specification. Among the several software metrics, functional size, or function points, has been one of the most favored ones in the literature. Although it is recognized and applied in practice since introduced in 1979, Functional Size Measurement (FSM) has not been as widely practiced in software community as expected. The problems with FSM method structures and practices have been considered to be a major factor to explain the situation. This chapter presents a review and analysis of FSM approaches that propose solutions to the problems with FSM via formalization of FSM practices.

Chapter 11. Cognitive Complexity Measures: An Analysis

A recent type of software measurement that is adopted from cognitive informatics is the cognitive complexity measure is presented. Unlike the computational complexity, that deals with the algorithm complexity and evaluates time and memory needed to execute a program, cognitive complexity refers to the human effort needed to perform a task or the difficulty experienced in understanding the code. In recent years, several cognitive complexity measures based on CI have been proposed with varying advantages and disadvantages. This chapter presents a critical review on existing cognitive complexity measures and a comparative study based on some selected attributes of these measures.

Chapter 12. Introducing Agility into Plan-Based Assessments

This chapter addresses the question of how to integrate agile practices with traditional plan-driven software process assessments. Assessments are critical within the organizations as they facilitate improvements in an organisation’s software development and management processes by characterizing
the current state of its software process and providing findings and recommendations to facilitate any improvement. Incorporating agile methods into software process assessments provides cost-effective and lightweight assessments that are highly desirable, especially for small to medium sized enterprises. The authors introduce a method, namely Agile Hybrid Assessment Method for Automotive industry (AHAA), to enable the organizations to combine CMMI, agile practices, and Automotive SPICE™ approaches together. The method is also implemented and validated in the scope of two SMEs in Ireland.

Chapter 13. Software Development Governance: A Case Study for Tools Integration

This chapter addresses software development governance that is considered as the application of “governance” in software engineering in order to better manage the individual projects, as well as the organizations involved in many interrelated projects. It is an interdisciplinary perspective that adopts the principles from the field of Transaction Cost Economics. This chapter analyzes the organizations that aim to adapt the right governance structure for their projects based on their characteristics and the tools to be aligned with this structure. A novel Dependency Structure Matrix (DSM) is presented to utilize the mapping between governance structures and software modules and is demonstrated with a successful integration using three broadly available tools in the software industry.

Chapter 14. A Software Cost Model to Assess Productivity Impact of a Model-Driven Technique in Developing Domain-Specific Design Tools

As model-driven development techniques become widespread, the design tools that utilize domain-specific models to design and generate software have also been a major prerequisite. The major predicament with these techniques is the complexity imposed when manually developing these domain-specific design tools. There is also a difficulty raised while integrating these design tools with model validation tools and code generators. This chapter presents a model-driven technique and its supporting model-driven environment to automate the tool development and integration. A formal parametric model is also proposed that allows evaluating the productivity impact in generating and rapidly integrating design tools.

Chapter 15. Model-Driven Development of Multi-Core Embedded Software

This chapter focuses on the development of multi-core embedded software. As multi-core processors are becoming prevalent rapidly in personal computing and embedded systems, it also becomes quite critical that the software to operate on a multi-core architecture is specifically designed; however, current programming environments for multi-core processor based systems are still quite immature and lack efficient tools. This chapter proposes a new framework called VERTAF/Multi-Core (VMC) and shows how software code can be automatically generated from high-level models of multi-core embedded systems. VMC aims to minimize the effort for developing embedded software for multi-core processors and to solve several of the issues related to model-driven development for multi-core embedded systems.
Chapter 16. Analyzing Concurrent Programs Title for Potential Programming Errors

Concurrency is the major point of focus in parallel architectures as it introduces more fragility for programming errors than sequential programs. As multi-core/multi-processor hardware becomes more prevalent, detecting concurrency errors effectively and efficiently has become a research focus of software engineering in the recent years. This chapter presents the major concurrent programming models including multithreaded programming on shared memory and message passing programming on distributed memory and reviews the state-of-the-art approaches on detecting concurrency errors.

Ali H. Doğru  
Middle East Technical University, Turkey

Veli Biçer  
FZI Research Center for Information Technology, Germany

ENDNOTES

1  http://www.projectsmart.co.uk/docs/chaos-report.pdf