Preface

Software engineering is a term that has a very broad definition. This process includes the logical design of a system; the development of prototypes, the automated generation of computer code for the system; the testing, validation and benchmarking of the code and the final implementation of the system. Once a new system is up and running, the software engineering process is used to maintain the system, evaluate its operation, keep track of new versions and refactor and/or reuse the code for other projects.

Over the past 30 years the discipline of software engineering has grown. In some cases, a specific programming paradigm, such as object-oriented, evolved into a broad discipline encompassing design and programming processes, tools and techniques. Several universities offer degrees as well as courses in software engineering. Standards for software engineering have been incorporated and formalized in England, Canada, Australia and the United States. Additionally, software engineering has received recognition from licensing and standards boards such as the Association of Computing Machinery (ACM) Institute of Electrical Engineering (IEEE), ISO 9000 and the Institute for Certification of Computing Professionals (ICCP).

Although many current design practices are focused on object-oriented techniques, this does not limit us to using object-oriented languages. It is quite possible to adopt the methods whether one writes in Fortran, C++ or writes scripts in Perl. In recent times the concept of software engineering has expanded to include not only code generation and system design, but a set of standards and methods that the software engineer should practice.

The practice of software engineering rightfully begins in the requirements phase of any system project, where the problem to be solved is well defined. Once this is captured, the design phase starts. In an effort to avoid the problem of "reinventing the wheel" a good designer decides what methods and patterns can be drawn from the existing software engineering "body of knowledge." Reusable generic design and code is only one advantage that has been realized today as the libraries of functions, patterns, and frameworks continue to grow.

Automated support for the application and integration of these reusable units with newly defined designs and modules using a Computer Aided Software Engineering (CASE) tool has created a new lexicon in this field. "Lower CASE" tools now refer to code generation while "higher CASE" tools are those tools used in the

construction and diagramming of proposed computer systems. There have recently been proposals to integrate these two capabilities into a single tool, so that once a system is proposed and analyzed using standard tools, such as Data Flow Diagrams (DFD), Entity Relationship Diagrams, and Unified Modeling Language (UML), this information is passed to another module of the tool to generate code consistent with these diagrams.

As previously mentioned in this preface, during the past 30 years a generalized body of knowledge about design as other aspects of software engineering processes has emerged with some generally accepted standards. The reader should refer to the "Guide to Software Engineering Body of Knowledge; SWEBOK" from IEEE Computer Society for an excellent description of this body of common knowledge and standards.

This book begins with a discussion of software patterns that are used to facilitate the reuse of object-oriented designs. While most CASE tools support the use of UML to extract the design from the software engineer and to assist in the development, most do not provide assistance in the integration and code generation of software patterns. In this chapter, the authors analyze the Iterator software pattern for the semantics that would be used in a CASE design tool to help the software engineer integrate this pattern into a design and then generate some of the code needed to implement the pattern. This work is based on semantic data modeling techniques that were previously proposed for the design of active databases.

The next chapter introduces a theoretical frame for processes definition validation in workflow processes with temporal restrictions. Workflow Interface 1 provides the process definition of the Work Flow Reference Model. This interface combines PNwC to provide the formalization and verification of systems based on the Petri Net theory with an extension. This extension allows the specification of temporal requirements via clock specification, using temporal invariants for the places and temporal conditions in the transitions. This chapter presents a technique to validate the process definition (PD) using Petri Nets with Clocks (PNwC). The algorithm for the analysis of a PNwC allows for the correction of errors in the modeling of the time variable. The algorithm generates information about temporal unreachable states and process deadlocks with temporal blocks. It also corrects activity invariants and transition conditions.

The third chapter identifies the key aspects of software engineering and systems engineering in an effort to highlight areas of consensus and conflict. The goal is to support current efforts by practitioners and academics in both disciplines to redefine their professions and bodies of knowledge. By using the Software Engineering Institute's Capability Maturity Model-Integrated (CMMISM) project, which combines best practices from the systems and software engineering disciplines, it can be shown that significant points of agreement and consensus are evident. Nevertheless, valid objections to such integration remain as areas of conflict. It is hoped that this chapter will provide an opportunity for these two communities to resolve unnecessary differences in terminology and methodologies that are reflected

in their differing perspectives and entrenched in their organizational cultures.

Historically the approach to software engineering has been based on a search for an optimal (ideal) methodology— the identification and application of a set of processes, methods and tools that can consistently and predictably lead to software development success. The fourth chapter presents the basis for pursuing a more flexible, adaptive approach. Less methodical techniques under a variety of names take what is described as a contingency-oriented approach. Because of the limitations in the nature of methodology, the high failure rate in software development, the need to develop methodology within an environmental context and the pressures of fast-paced "E" development, the authors argue that further exploration and definition of an adaptive, contingency-based approach to methodology is justified.

Chapter V challenges the established wisdom with respect to use cases. Use cases are classically elaborate to capture the functional requirements of the system by directly identifying objects, methods and data. Several authors of system analysis and design books advocate this approach. However the research reported in this paper indicates that there are better constructs for modeling use cases, at least initially. Further objects are not a particularly good medium for discussing requirements with users. This paper rehearses the arguments leading up to these conclusions and identifies some implications of these conclusions.

The theme of system development is continued in Chapter VI. Using the RAISE specification development process, a variety of components and infrastructures are built. These components are not independent and are related to each other, when the authors specify different systems into the same infrastructure. The RAISE method is based on the idea that software development is a stepwise, evolutionary process of applying semantics-preserving transitions. Reuse is crucially impacted in all the stages of the development, but there is no explicit reference to the specification of reusability in this development process. This chapter presents a rigorous process for reusability using RSL (RAISE Specification Language) components. The authors provide the mechanism to select a reusable component in order to guide RAISE developers in the software specification and construction process.

The section on system development concludes with Chapter VII. This chapter introduces the Functional and Object-Oriented Methodology (FOOM) . This is an integrated methodology for information systems analysis and design that combines two essential software-engineering paradigms: the functional/data approach (or process-oriented) and the object-oriented (OO) approach. FOOM has been applied to a variety of domains. This chapter presents the application of the methodology to the specification of the "IFIP Conference" system with focus on the analysis and design phases. The FOOM analysis phase includes data modeling and functional analysis activities, and produces an initial Class Diagram and a hierarchy of OO Data Flow Diagrams (OO-DFDs). The products of the design phase include: (a) a complete class diagram; (b) object classes for the menus, forms and reports and (c) a behavior schema, which consists of detailed descriptions of the methods

and the application transactions expressed in pseudocode and message diagrams.

Section II discusses methods to evaluate and manage the system development process. Chapter VIII presents a comprehensive quantitative management model for information technology. This methodology is assessment based and can be easily implemented without imposing an unacceptable organizational change. It supplies detailed information about the functioning of processes that allows managers to both effectively oversee operations and assess their prospective and ongoing execution risks. This offers a consistent risk reward evaluation.

Continuing with the theme of measurement and risk assessment Chapter IX describes the foundation and properties of specific object-oriented software measures. Many measures for object-oriented applications have been constructed and tested in development environments. However, the process of defining new measures is still alive. The reason for this lies in the difficulties associated with understanding and maintaining object-oriented applications. It is still difficult to relate the measures to the phenomena that need to be improved. Do current measurements indicate problems in reliability, maintenance or the unreasonable complexity of some portions of the application?

In order to reduce the complexity of software, new development methodologies and tools are being introduced. The authors talk about a new approach to development called separation of concern. Tools, such as Aspect/J or Hyper/J, facilitate the development process, but there does not seem to be a sound metrics suite to measure complexity and efficiency of applications developed and coded with Aspect/J or Hyper/J. In this chapter, the authors attempt to review the current research into object-oriented software metrics and suggest theoretical framework for complexity estimation and ranking of compositional units in object-oriented applications developed with Hyper/J.

Chapter X concludes the managing projects section by introducing a novel notion of temporal interaction diagrams that can be used for testing and evaluating distributed and parallel programming. An interaction diagram is a graphic view of computation processes and communication between different entities in distributed and parallel applications. It can be used for the specification, implementation and testing of interaction policies in distributed and parallel systems. Expressing interaction diagrams in a linear form, known as fragmentation, facilitates automation of design and testing of such systems. Existing interaction diagram formalisms lack the flexibility and capability of describing more general temporal order constraints. They only support rigid temporal order, hence have limited semantic expressiveness. The authors propose an improved interaction diagram formalism in which more general temporal constraints can be expressed. This enables the capture of multiple valid interaction sequences using a single interaction diagram.

Section III discusses specific applications and implementations that are best solved by the principles of software engineering. Chapter XI begins this section with relevant security issues that must be considered in any software implementation. While academicians and industry practitioners have long recognized the need

for securing information systems and computer architectures, there has recently been a heightened awareness of information technology (IT) management on computer-related security issues. IT managers are increasingly worried about possible attacks on computer facilities and software, especially for mission critical software. There are many dimensions to providing a secure computing environment for an organization, including computer viruses, Trojan horses, unauthorized accesses and intrusions and thefts to infrastructure. This complexity and multidimensional nature of establishing computer security requires that the problem be tackled on many fronts simultaneously. Research in the area of information systems security has traditionally focused on architecture, infrastructure and systems level security. Emerging literature on application-level security, while providing useful paradigms, remain isolated and disparate. The current study focuses on single, albeit an important, dimension of providing a safe and secure computing environment — application-software security.

The book progresses to a specific proposal for learning systems. Chapter XII presents a project proposal for future work utilizing software engineering concepts to produce learning processes in cognitive systems. This project outlines a number of directions in the fields of systems engineering, machine learning, knowledge engineering and profile theory that lead to the development of formal methods for the modeling and engineering of learning systems. This chapter describes a framework for formalization and engineering of the cognitive processes and is based on applications of computational methods. The work proposes the studies of cognitive processes in software development process, and considers a cognitive system as a multi-agent system of human cognitive agents. It is important to note that this framework can be applied to different types of learning systems. There are various techniques from different theories (e.g., system theory, quantum theory, neural networks) that can be used for the description of cognitive systems, which, in turn, can be represented by different types of cognitive agents.

Web-based applications are highlighted by Chapter XIII. Global competition among today's enterprises forces their business processes to evolve constantly, leading to changes in corresponding Web-based application systems. Most existing approaches that extend traditional software engineering to develop Web-based application systems are based on OO methods. Such methods emphasize modeling individual object behaviors rather than system behavior. This chapter proposes the Business Process-Based Methodology (BPBM) for developing such systems. It uses a business process as a unified conceptual framework for analyzing relationships between a business process and associated business objects, identifying business activities and designing OO components called business components. The authors propose measures for coupling and cohesion measurement in order to ensure that these business components enable the potential reusability. These business components can more clearly represent semantic system behaviors than linkages of individual object behaviors. A change made to one business process impacts some encapsulated atomic components within the respective business component without

affecting other parts of the system. A business component is divided into parts suitable for implementation of multi-tier Web-based application systems.

Geographic Information Systems (GIS) are presented in Chapter XIV. This chapter introduces an OO methodology for GIS development. It argues that a COTS-based development methodology combined with the UML can be extended to support the spatio-temporal peculiarities that characterize GIS applications. The authors suggest that by typifying both enterprises and developments, and, with a thorough knowledge of the software component granularity in the GIS domain, it will be possible to extend and adapt the proposed COTS-based methodologies to cover the full lifecycle. Moreover, some recommendations are outlined to translate the methodology to the commercial iCASE Rational Suite Enterprise and its relationships with tool kits proposed by some GIS COTS vendors.

Chapter XIV makes the claim of improved efficiency and reliability of networking technology, providing a framework for service discovery where clients connect to services over the network. It is based on a comparison of the client's requirements with the advertised capabilities of those services. Many service directory technologies exist to provide this middleware functionality, each with its own default set of service attributes that may be used for comparison and its own default search algorithms. Because the most expressive search ability might not be as important as robustness for directory services, the search algorithms provided are usually limited when compared to a service devoted entirely to intelligent service discovery.

To address the above problems, the authors propose a framework of intelligent service discovery running alongside a service directory that allows the search service to have a range of search algorithms available. The most appropriate algorithm may be chosen for a search according to the data types found in the search criteria. A specific implementation of this framework is presented as a Jini service, using a constraint satisfaction problem solving architecture that allows different algorithms to be used as library components.

Although component-based development (CBD) platforms and technologies, such as CORBA, COM+/.NET and enterprise Java Beans (EJB) are now de facto standards for implementation and deployment of complex enterprise distributed systems, according to the authors of Chapter XVI, the full benefit of the component way of thinking has not been gained. Current CBD approaches and methods treat components mainly as binary-code implementation packages or as larger grained business objects in system analysis and design. Little attention has been paid to the potential of the component way of thinking in filling the gap between business and IT issues. This chapter proposes a service-based approach to the component concept representing the point of convergence of business and technology concerns. The approach defines components as the main building blocks of business-driven service-based system architecture that provides effective business IT alignment.

The book now focuses its attention on specific issues of software engineering as applied to telecommunications networks. Chapter XVII describes the design of

a narrowband lowpass finite impulse response (FIR) filter using a small number of multipliers per output sample (MPS). The method is based on the use of a frequency-improved recursive running sum (RRS) called the sharpening RRS filter and the interpolated finite impulse response (IFIR) structure. The filter sharpening technique uses multiple copies of the same filter according to an amplitude change function (ACF), which maps a transfer function before sharpening to a desired form after sharpening. Three ACFs are used in the design as illustrated in examples contained in this chapter.

The book closes with Chapter XVIII, which describes another telecommunication application. This chapter presents the design of narrowband highpass linear-phase FIR filters using the sharpening RRS filter and the IFIR structure. The novelty of this technique is based on the use of a sharpening RRS filter as an image suppressor in the IFIR structure. In this way the total number of multiplications per output sample is considerably reduced.

The purpose of this book is to introduce new and original work from around the world that we believe expands the body of common knowledge in software engineering. The order of this book attempts to tell a story, beginning with the software process, including reusable code and specific design methodologies and the methods associated with this formalized structure. The book then proceeds to chapters that propose models to measure the system analysis and design process and to direct the successful development of computer systems. The chapters then progress to the next step in any system project — the implementation phase. This section includes various aspects of using and integrating the engineered software into a computer system. Its chapters address security and systems capable of learning. The book then concludes with specific examples of Web-based and telecommunication applications.