Using Aspect-Oriented Programming for Trustworthy Software Development

VLADIMIR O. SAFONOV
Using Aspect-Oriented Programming for Trustworthy Software Development
Using Aspect-Oriented Programming for Trustworthy Software Development

VLADIMIR O. SAFONOVo
Contents

Preface ix

1. Introduction 1

1.1 The Role of Aspect-Oriented Programming in Trustworthiness / 1
1.2 Historical Background and Personal Experience / 3
1.3 Organization of the Book / 8

2. Trustworthy Computing, Software Engineering, and Computer Science 10

2.1 History of and Growing Need for TWC / 10
2.2 Microsoft’s TWC Initiative / 13
2.3 The Four Pillars of TWC / 15
  2.3.1 Security / 15
  2.3.2 Privacy / 23
  2.3.3 Reliability / 25
  2.3.4 Business Integrity / 27
2.4 Software Engineering Technologies and Tools for TWC / 30
2.5 TWC and .NET / 31
  2.5.1 .NET Overview / 31
  2.5.2 .NET Security / 34
  2.5.3 .NET and Reliability / 37
  2.5.4 .NET TWC Tools FxCop and Spec# / 39
CONTENTS

2.6 TWC and Java / 42
  2.6.1 Java Overview / 42
  2.6.2 Java Security / 44
  2.6.3 Java and Reliability / 46
  2.6.4 Java TWC Tools / 48

2.7 Summary / 49

3. Aspect-Oriented Programming and Aspect.NET / 50
  3.1 History of AOP / 50
  3.2 AOP Basics / 54
  3.3 AOP and Related Technologies and Tools / 66
    3.3.1 AspectJ and AspectWerkz / 66
    3.3.2 Other AOP Tools and Approaches to Separation of Concerns / 70

3.4 Pitfalls of AOP / 73
3.5 AOP for Java / 76
3.6 AOP for .NET / 79
3.7 Aspect.NET Principles and Architecture / 85
  3.7.1 Motivation and Key Ideas / 85
  3.7.2 Basic Concepts of AOP / 88
  3.7.3 Example / 91
  3.7.4 Representing Aspects by Custom Attributes / 92
  3.7.5 Example in Terms of Custom Attributes / 94
  3.7.6 Summary of Our Approach to AOP / 95
  3.7.7 Aspect.NET Architectural Principles / 97
  3.7.8 Syntax of AOP Metalanguage (Version 1.0) / 99
  3.7.9 Another Example / 101

3.8 Features and Use of Aspect.NET / 102
  3.8.1 Prerequisites for Using Aspect.NET 2.1 / 103
  3.8.2 Previous Releases of Aspect.NET and the Compatibility Mode / 103
  3.8.3 Aspect.NET Architecture / 104
  3.8.4 Case Study: Using the Aspect.NET Framework / 106
  3.8.5 Aspect.NET Framework Options / 113
  3.8.6 Aspect.NET.ML Metalanguage / 114
  3.8.7 Samples Included in the Aspect.NET 2.1 Release / 122
  3.8.8 Experience of Aspect.NET Use and User Feedback / 123

3.9 Summary
  3.9.1 AOP / 123
  3.9.2 Aspect.NET / 125
4. Principles and Application of AOP in TWC 128

4.1 AOP and TWC: Cooperation Rather Than Violation / 128
4.2 AOP for Security / 132
4.3 AOP for Error Handling / 136
4.4 AOP for Synchronization / 142
4.5 AOP for Trustworthy Multithreading- and Multicore-Based Applications / 145
4.6 AOP for Privacy / 149
4.7 AOP for Reliability / 153
  4.7.1 Using AOP to Make Implementation Reliable / 153
  4.7.2 Using AOP for Software Testing / 157
  4.7.3 Using AOP to Support Formal Specification and Verification Methods / 164
4.8 AOP for Business Integrity / 165
4.9 AOP for Design by Contract / 168
4.10 Using AOP via Aspect.NET to Improve Productivity and Reliability / 183
  4.10.1 Effort Estimation Using the COCOMO Model / 184
  4.10.2 Assessment of Aspect.NET Using the ICED-T Model / 187
  4.10.3 Assessment of Requirements of Aspect.NET Using the SQFD Model / 190
4.11 Application Efficiency and Performance Using AOP / 193
  4.11.1 Performance Measurement / 194
  4.11.2 Implementation Details and the Woven IL Code / 197
  4.11.3 Another Performance Measurement Example / 198
4.12 AOP and Agile Programming Approaches / 202
4.13 Summary / 206

5. Teaching TWC and AOP 208

5.1 The ERATO Teaching Paradigm and the SPBU.NET Project / 209
  5.1.1 The ERATO Teaching Paradigm / 209
  5.1.2 The SPBU.NET Project / 212
5.2 The T-ERATO Teaching Paradigm and the TrustSPBU.NET Project / 215
  5.2.1 The T-ERATO Teaching Paradigm / 215
  5.2.2 The TrustSPBU.NET Project / 217
5.3 Teaching Trustworthy Software Engineering, Including AOP / 220
  5.3.1 Structure of the Secure Software Engineering Course / 220
5.4 Teaching Trustworthy .NET and C# Programming / 238
   5.4.1 Structure of the Trustworthy .NET and C# Programming Course / 239
   5.4.2 Structure of the .NET and C# Programming Seminar / 249
5.5 Teaching Trustworthy Java Technology / 251
   5.5.1 Structure of the Java Course / 252
   5.5.2 Structure of the Java Seminar / 257
5.6 Teaching Trustworthy Operating Systems and Networking / 260
   5.6.1 Structure of the Trustworthy Operating Systems and Networking Course / 262
5.7 Teaching Trustworthy Compiler Development / 263
   5.7.1 Structure of the Trustworthy Compiler Development Course / 265
   5.7.2 Structure of the Compiler Development Seminar / 266
5.8 Summary / 268

6. Conclusion 269
   6.1 Summary of the Book / 269
   6.2 Perspectives: Integration of TWC, AOP, Formal Methods, and Knowledge Management / 271
      6.2.1 Application of Formal Methods to Trustworthy AOP / 272
      6.2.2 Smart Knowledge-Based AOP / 273
      6.2.3 AOP Specification and Design Support Tools / 273
      6.2.4 Trustworthy Reverse Engineering and Refactoring Tools for AOP / 274
      6.2.5 Aspect-Oriented Modeling on the Basis of Extended UML / 275

Appendix: Examples of Aspect.NET Aspects 276
   A.1 TestArgs Example / 276
   A.2 RetTest Example / 279
   A.3 RetTest2 Example / 282
   A.4 QuickSort Example / 284
   A.5 Matrix Example / 295

References 321

Index 329
This book is devoted to the basic concepts and generic relationships of two new software engineering and computer science areas: trustworthy computing (TWC) and aspect-oriented programming (AOP).

These two disciplines are now so popular, even "fashionable," that many software students and experts are looking for more information about them. Both TWC and AOP and, in particular, their relationship, have not yet been described sufficiently in the scientific literature up to now, and this gap needs to be filled. The reason for the current status lies in the fact that both AOP and TWC are still quite novel even though their foundations were laid long ago, and the worldwide software engineering community has not yet acquired enough experience in these areas. The book should be very helpful in this regard.

The main principle behind the book is that AOP, used properly, can be beneficial in trustworthy software development, due to the fact that the two are related generically. To analyze, demonstrate, and teach using AOP for TWC, I take typical TWC tasks, such as security checks, in and out conditions, and multithreaded safety and show how they can be implemented using AOP, since I believe that in terms of AOP, most TWC tasks are cross-cutting concerns and, consequently, can and should be implemented as aspects. In the text I describe my team’s latest results, analyzing my 30 years’ experience in research, development, and university teaching in the areas of software engineering and computer science, and my 15 years’ experience collaborating with leading global software companies: Microsoft Research, Sun Microsystems, and others.

In particular, I describe the results of two research and educational projects supported by Microsoft Research:
• **Aspect.NET** [1–6]: an AOP framework for the Microsoft.NET platform based on Microsoft Phoenix [7] and Microsoft Visual Studio.NET 2005. Aspect.NET is based on our approach to AOP as well as our AOP framework. Aspect.NET already has a number of users in 16 countries, including the United States, Canada, and other countries in the Americas, Europe, Asia, and the C.I.S. We hope the book will contribute to extending the Aspect.NET user community.

• **TrustSPBU.NET** [8]: a set of educational materials on advanced secure software engineering and trustworthy computing, Microsoft.NET and C#, compilers, software engineering and compiler development, and my related project, SPBU.NET [9], used as the foundation for TrustSPBU.NET. The curriculum materials of these two projects, available on Microsoft Developer’s Network Academic Alliance Curriculum Repository Web site, have already attracted the attention of both students and software experts.

For all examples of trustworthy software design and code included in the book, I use our Aspect.NET framework. I consider the basics of Aspect.NET architecture, its advantages compared to other AOP tools, its functionality, and examples of trustworthy application development using Aspect.NET. The book is not limited to Microsoft technologies, although we do appreciate using such advanced toolkits as Visual Studio.NET and Phoenix. We consider general principles and other software technologies and tools applicable to using AOP for trustworthy software development, such as Java and AspectJ [10], based on Java, currently the most widely used AOP instrument.

Chapter 5 is devoted to teaching, but actually, the style used to present all the material in the book is based on the ERATO teaching paradigm [9], on which I have based my university teaching for many years. ERATO is an acronym for experience, retrospective, analysis, theory, oncoming perspectives. Erato is the name of the muse of romantic poetry in ancient Greek mythology. The ERATO teaching paradigm can be summarized as follows:

• **Experience**: describing my long-term commercial and research software project experience in my courses. In particular, in 1992–2002 I led St. Petersburg Sun projects in the compiler development and Java technology areas. In 2002 I started working with Microsoft Research on the Aspect.NET project, in 2003 on the Phoenix compiler development tool, in 2004 on SPBU.NET, and in 2006 on TrustSPBU.NET educational projects. Such types of things are of deep interest to students since they can judge how closely coupled academic learning and teaching activity can be to advanced research and tools and working with leading companies, so it helps to better illustrate concepts and principles to be taught. They can participate personally in our projects to get their own experience.
• **Retrospective:** considering the historical background of each topic being taught since its early origin, for deeper understanding of fundamental concepts by the students. For example, I consider the concepts of concurrency and multithreading since Dijkstra’s 1960s pioneering work on semaphores; generics since Liskov’s CLU language parametrized types in early 1970s (rather than since 2004–2005, when generics were implemented in Java 1.5 and C# 2.0).

• **Analysis:** making critical and comparative analysis of the most important related and mutually influenced concepts and technologies when teaching them. For example, I analyze the Microsoft.NET platform, compared to the competitive Java platform, and explain the fundamental reasons why .NET is more general and open-style. On the other hand, I show to students that .NET technologies have a backward influence on Java. I believe that in this way students can better understand the dialectic nature of software engineering.

• **Theory:** formulating and explaining the essence of theoretical definitions, justifications, known theorems, and issues relevant to the topic being taught. In particular, when teaching the concept of data type, I make a review of the techniques of formal specifications of abstract data types: Hoare’s theoretical papers on data types published in the 1960s and 1970s; papers by Scott on type theory; and pioneering papers on initial and final algebra semantics of abstract data types by the ADJ group (1970s), resulting in algebraic data type specification languages OBJ and SDL.

• **Oncoming perspectives:** explaining the vision of future progress in the topic being taught by a variety of software experts.

The book is targeted primarily at undergraduate and graduate students who would like to study TWC and AOP, but it will also be useful for software managers, computer scientists, software engineers, and university teachers in the area, especially those working with and teaching Microsoft.NET. For readers who want to learn the basics of trustworthy computing and its application to modern software development platforms such as .NET and Java, Chapter 2 will be appropriate. Chapter 3 focuses on readers interested primarily in aspect-oriented programming and Aspect.NET. Those specifically interested in applying AOP to develop trustworthy software are directed to Chapter 4. Chapter 5 will be of most interest to those who wish to teach TWC and AOP and related areas of software engineering.

For more information, the reader is directed to the book and to the Aspect.NET Web site: www.aspectdotnet.org. The site contains all the examples used in the book, other material selected from the book, the Aspect.NET framework with documentation, and data on other publications related to Aspect.NET.

Please send your questions, remarks, suggestions, and proposals regarding Aspect.NET or my book directly to my e-mail address: v_o_safonov@mail.ru.
ACKNOWLEDGMENTS

First, I would like to thank my colleagues from Microsoft: Igor Agamirzian, Marco Combetto, Peter Drayton, Van Eden, Kang Su Gatlin, Alexander Gavrilov, Alexander Gil (who was my doctoral student), John Larcheveque, John Lefor, Mark Lewin, Vassily Lutsarev, Shahrokh Mortazavi, John Spencer, Andrey Terekhov, Damien Watkins, Yan Xu, Romualds Zdebskis, and many others. They have provided a lot of support to me and my team.

Thanks also to my teachers from a variety of universities and companies whose ideas, papers, education, and attention have had a great influence on my work, results, professional views, and interests: Professors Alfred Aho (Columbia University), Igor Bratchikov, Boris Martynenko and Joseph Romanovsky (St. Petersburg University), Gregory Tseytin (St. Petersburg University, now with IBM), and Niklaus Wirth (ETH Zürich).

Special thanks to Professor Gregor Kiczales, the inventor of AOP, for fruitful e-mail discussions that helped me to understand the essence of his approach more deeply.

I would like to memorialize two people whose papers inspired my professional work: corresponding member of the Russian Academy of Sciences Professor Svyatoslav Lavrov, my scientific advisor, the developer of the first ALGOL compiler in the Soviet Union in the early 1960s, and the author of many books and projects well known in our country; and Professor Adolph Fouxman of Rostov University, the author of the vertical cut technology (1979), predecessor of aspect-oriented programming, and initiator of pioneering projects on automated program synthesis, whose software engineering papers of 1970s look quite contemporary.

Finally, I would like to thank deeply my Aspect.NET implementation team: my doctoral students Dmitry Grigoryev, the developer of the Aspect.NET weaver using Microsoft Phoenix [7]; Mikhail Gratchev, the developer of the Aspect.NET framework and its integration in Visual Studio.NET 2005 and Aspect.NET installer; and Alexander Maslennikov, the developer of the Aspect.NET.ML metalanguage converter to C#. They have not only developed several releases of Aspect.NET but have also proposed and implemented a lot of their own bright ideas to improve and enhance Aspect.NET architecture and functionality. The results of their work should be considered an inherent part of my book. Also, thanks to my graduate students Oleg Romashin and Ruslan Mukhanov, who helped me to develop AspectRotor, a version of Aspect.NET for SSCLI (Rotor), and to Anna Kogay, who developed a number of useful aspect examples of using Aspect.NET for the design-by-contract technology used in the book.

Last but not least, my thanks to all Aspect.NET users and supporters.
In this introductory chapter we explain why aspect-oriented programming is so closely related to trustworthy software development, and review the organization of the book in detail.

1.1 THE ROLE OF ASPECT-ORIENTED PROGRAMMING IN TRUSTWORTHINESS

Each software product is expected by its users to be trustworthy. But the concept of software trustworthiness consists of many parts and aspects, as we’ll see later. Intuitively, this concept has evolved since the early days of programming. Thanks to Microsoft’s announcement in 2002 to follow and support the trustworthy computing initiative, software experts are paying much more attention to the task of making trustworthy computing a more systematic discipline.

It is very important to understand that viewing particular software as trustworthy or nontrustworthy generally evolves over time and may also depend on the environment and the target platform. Software considered to be trustworthy at one moment or in one environment may demonstrate nontrustworthy behavior at another moment or after porting to another platform or environment. Let’s consider a real example that occurred in the 1980s with
one of our university’s mathematical packages written in FORTRAN that we were porting from IBM 360 mainframes to Soviet Elbrus [11] computers with tagged architecture. The package seemed to work fine and to deliver trustworthy results for a few years while running on an IBM 360. During the first run of the package on Elbrus, an interrupt occurred and a runtime error (“invalid operand”) was detected by hardware. It appeared that some noninitialized variable was used in the package. It is intriguing that the package worked reasonably on an IBM 360, but porting to a new, more secure hardware architecture obviously contributed to making the software more trustworthy, by detecting and fixing the bug immediately.

As another example, let’s consider a library, part of a legacy code that works well and is trustworthy in a single-threaded environment but needs to be updated to become multithreaded (MT) safe. The library may contain static variables and other implementation specifics that can preclude its trustworthiness in a multithreaded environment, so to achieve MT safety, its code needs to be updated systematically by, at a minimum, adding calls to synchronization primitives (semaphores, mutexes, etc.) before and after the operations that retrieve or update global resources shared by the library’s functions.

There are many more other situations in which the existing code should be updated to become more trustworthy. Due to the evolving nature of hardware and software platforms, networking and operating system (OS) environments, and the tasks to be solved by software, most software products cannot be developed to be trustworthy “once and for all.” So a typical task in modern software development is to update software to improve its trustworthiness in some sense or in some relation. Such a software update should be made safely and systematically, according to an explicit and consistent plan, using special tools adequate to such a task, preferably with the help of formal methods such as specification and verification.

It is very important to realize that the task of updating software to attain more trustworthiness generically has a “cross-cutting” nature. For example, to insert synchronizing actions into library code modules, it is necessary to locate in its code the scattered fragments responsible for updating or retrieving global shared resources, and then insert synchronizing actions before and after them. In other words, what is needed is to cross-cut the existing code for the purpose of adding tangled (but logically related) fragments responsible for synchronization.

All of the above can be achieved using aspect-oriented programming (AOP) [12]. AOP is a new discipline targeted to systematic software updates using a new type of modular units: aspects, whose code fragments are woven into the target application according to explicit weaving rules provided in the aspect definition. The main goal of AOP is to handle cross-cutting concerns [13]: ideas, considerations, or algorithms whose implementation, inherently, by its nature, cannot be implemented by a generalized procedure: a new function, class, method, or hierarchy of such related entities. Implementing a cross-cutting concern requires weaving (injecting) tangled code fragments into existing code
modules. Typical cross-cutting concerns formulated by the classicists of AOP are security, synchronization, and logging—all related to trustworthy computing. So, generally speaking, we can state that typical cross-cutting concerns are related to trustworthy software development. In this book you’ll find practical confirmation of this important principle, and practical recipes for how to use AOP for trustworthy software development, based on many examples. More precise definitions of trustworthy computing (TWC) are provided in Chapter 2, and of AOP, in Chapter 3.

1.2 HISTORICAL BACKGROUND AND PERSONAL EXPERIENCE

The foundation of trustworthy software development was laid out in the 1960s and 1970s, when programming became a systematic discipline, due to pioneering work by the classicists of structured programming (C. Boehm, J. Jacopini, E. Dijkstra, N. Wirth), modular programming (D. Parnas, G. Myers), and abstract data types (C. A. R. Hoare, F. Morris, D. Scott, J. Goguen, et al.).

In the late 1960s, the problem of “bowl-of-spaghetti” programs was investigated [14,15]. The criticism of such programs was that the code “overpatched” with goto statements—added quickly to patch existing code. As a result, programs became nontrustworthy and even unreadable. To correct this situation, the discipline of structured programming [16,17] was proposed. Structured programming is based on using a limited set of syntactically and semantically clean constructs: a succession of statements, if and while statements, without the use of goto, can therefore be considered the first attempt to make the code and the process of its development more trustworthy. An inherent part of structured programming is stepwise refinement: an incremental top-down software design and implementation technique using Pascal-like pseudocode to design each node of the program structure tree. Structured programming and stepwise refinement have played a historical role and are still being used in many projects, especially for rapid prototyping. Structured programming has many advantages: It provides a comfortable way to develop a software prototype quickly; it makes it possible to design and implement parts of the code in parallel; it enables clear control structure in the resulting code (which, in this sense, appears to get more trustworthy). However, structured programming has some limitations and shortcomings: It is suitable for statements (executable constructs of the code) but unsuitable for use in designing and developing definitions and declarations; it is not comfortable to use to make changes in a program; it is not well suited to error and exception handling; and it may cause the development of low-quality code since it can target highly skilled developers to design high-level pseudocode and have nonexperienced programmers make all implementations (i.e., actually produce the resulting code for the software product).

Modular programming [18,19] is another important step in trustworthy software development. It is intended for structural decomposition of a program
INTRODUCTION

into logically independent parts: *modules*. It is very important to understand that from a modular programming viewpoint, parts of the module interface are not only its name, arguments, and result types, but also a list of possible *exceptional conditions* of its possible abnormal termination. What is especially important for TWC is that trustworthy checks and exceptional conditions can be tied to modules. In terms of modular programming, TWC principles can be formulated as follows: *Each module and all module relationships should be trustworthy.* For that purpose, the module should self-check its pre- and post-conditions, handle all internal exceptions, and explicitly export its public exceptions to be processed by its users. Ideally although it has not yet been achieved in most software projects), the trustworthiness of the module should be proved by its formal verification, based on some form of its formal specification. The main shortcoming of classical modular programming [19] is lack of support for *cross-cutting concerns*. In traditional meaning, a module is visible and accessible from the rest of the application via its interface only. Implementation of the module is hidden, and it is not possible to change it from the outside in any way: in particular, to inject any new code into its implementation. However, as we have seen, the tasks of trustworthy software development require injecting new code (e.g., for synchronization or for security checks) into the implementation of existing modules. From a conventional modular programming viewpoint, it may be considered as a violation of modular programming principles. But since support for systematic code injections is a practical need of trustworthy software development, the concepts and views of modular programming have been augmented to justify systematic ways to inject or change tangled code. AOP is one approach to achieving this goal.

*Abstract data types* (ADTs) [20,21] represent the third classical approach to making software development more trustworthy, based on defining a set of operations on some data structure or collection (tree, list, graph, stack, etc.) as a new abstract data type whose implementation is *encapsulated* (hidden) inside the ADT definition. From the most general viewpoint, an inherent part of an ADT is its *formal specification*, which allows us formally to *verify* the correctness of its implementation (i.e., to prove that the implementation of the ADT corresponds to its formal specifications). Unfortunately, most existing languages, tools, and environments that include ADT mechanisms do not support formal specification and verification, which makes applying the discipline and concepts of ADT less trustworthy than expected. However, two very important ideas that the concept of ADT has brought to trustworthy software development are encapsulation of the concrete representation of the data type and exclusive use of *abstract operations* (implemented via methods, functions, macros, etc.) to handle objects of the data type without “intrusion” into its representation by straightforward use of its concrete elements.

So, to summarize, the concepts of structured programming, modular programming, and abstract data types, together with the related formal methods, contributed a lot to trustworthy software development of executable parts of programs: statements (structured programming), program architecture by its
decomposition into modules (modular programming), and operations on complicated data types and structures (abstract date types). But all of these approaches lack the support of cross-cutting concerns—hence the important role of AOP, which provides such support.

As for object-oriented programming (OOP), it has been playing an outstanding role in the rapid development of software, but its key principles and mechanisms, inheritance and class hierarchy, are not always trustworthy. As shown in analytical and critical papers on OOP, it has many pitfalls [22]. The most serious of them is conceptual explosion, an immediate consequence of implementation inheritance. By inheriting an exponentially growing number of classes and methods, an OOP programmer is likely to develop nontrustworthy and unreadable code, since it uses hundreds of implicitly inherited features implemented by other programmers, whose semantics is often unclear and not well documented. Nevertheless, OOP constructs are comfortable for tying various features to classes and methods: in particular, security checks and other TWC support code.

As for security and information protection, one of the earliest papers on this subject seems to be that if Saltzer and Schroeder [23], dating back over 30 years, which considers basic concepts and principles related to information protection and access rights to information. Trustworthy computing issues related to sharing and scheduling common resources were studied in the 1960s and 1970s, primarily in relation to operating systems. New types of security issues arose later, due to evolution of networking, the Internet, and the Web.

My own experience of trustworthy software development began in the 1970s. It happened that the first hardware and OS platform I used for developing research and commercial software projects was not IBM or PDP, but Elbrus [11], a Soviet family of computers whose architecture was inspired by the ideas of Iliffe’s paper [24] and Burroughs 5000/5500/6700/7700 hardware architecture. Elbrus software, including the operating system, compilers, and application packages, was developed from scratch by thousands of highly skilled Soviet software engineers in a specialized high-level language. I was the project lead of the St. Petersburg University team, which in a few years developed Pascal, CLU, Modula-2 compilers, and FORTH-83, SNOBOL-4, and REFAL [25] interpreters for Elbrus. These projects became my first school of trustworthy computing. Elbrus architecture was based on tags, hardware-supported runtime data-type information attached to every word of memory. Another important principle of Elbrus was its support of basic features of high-level languages and their implementation in hardware. There was no assembler language in Elbrus in the traditional meaning of the term. Instead, all its system software, including the OS and compilers, was developed in EL-76 [26], a dynamically typed high-level language whose syntax was close to that of ALGOL 68. It is interesting now, from a modern TWC viewpoint and with our .NET experience, to analyze Elbrus architecture. At first, what was helpful for TWC was its tagged architecture, a kind of dynamic typing. An example of how it helps has already been given. Hardware-supported dynamic
typing helps to detect the use of noninitialized variables, due to the special
empty tag representing noninitialized data detected by each instruction (e.g.,
arithmetic operation) and causing an interrupt. A tagging mechanism prevents
the spread of nontrustworthy results over the program execution data flow. It
also helps to protect memory: a special descriptor tag marks each address word
that points to an array, and contains its initial address and size. So no violation
of array bounds, no buffer overrun [27] security flaws, and no C-like address
arithmetic operations (which often cause TWC issues) are possible with tagged
architecture. So such architecture looks more trustworthy and more secure
than the traditional ×86 architecture. However, tagged architecture requires
substantial overhead for analyzing the tags of the operands by hardware
during execution of each instruction. For that reason it is not often used in
modern hardware.

When I switched from Elbrus to using an IBM PC with MS-DOS and Turbo
Pascal in the late 1980s, I was enjoying a very comfortable and trustworthy
integrated development environment (IDE). I was surprised, however, that
the IDE did not, by default, detect bugs such as array indexing out of bounds,
which is especially dangerous since it may cause buffer overrun flaws. To detect
indexing out of bounds, the user needed explicitly to switch on a special code
generation option that generated much less efficient code, whereas tagged
architecture would catch indexing out of bounds at once. So, in a sense, a new
IDE working on a traditional hardware platform appeared to be less trustwor-
thy than “good old” tagged architecture. For me, that was a useful lesson in
trustworthy computing.

Much later, in the mid-1990s, due to my work with Sun, I became acquainted
with Java, and in the early 2000s, due to collaboration with Microsoft Research,
started working with Microsoft.NET. I was pleasantly surprised to discover in
those new software technologies a “new incarnation” of the ideas of dynamic
typing that became the basis of Java and .NET trustworthiness, in the form of
managed code execution using metadata. Dynamic typing in Java and .NET
provide a basis for security checks, runtime type checking, array bounds check-
ing, and a lot of other elements of trustworthy code execution. It would not
be realistic to use hardware-supported tags for TWC purposes in modern
computing, due to modern software requirements of scalability, flexibility,
cross-platform nature, and network- and Web-awareness. Instead, the two
related technologies—common intermediate code (Java bytecode and MSIL)
based on postfix notation, and efficient just-in-time compilation of that inter-
mediate code to native code—are used in Java and .NET to support type-safe
execution.

I became acquainted with the principles and concepts of AOP in 2001
through an article by Elrad et al. [28]. Reading the article made me realize
that like many other experienced software developers I had intuitively used
and developed principles similar to AOP throughout my professional life,
although without explicit use of AOP terminology or adequate tools. For
example, a typical everyday software engineering task is to correct or extend
some functionality in an existing application: either your own or that of other developers (which can be much more difficult). To do that you should first locate the implementation of the functionality being considered within the application code and then modify and enhance it. Typically, implementation of some functionality consists of a related group of modules (class hierarchies, libraries of functions, etc.) and a set of scattered code fragments (i.e., definitions and calls of modules) in different parts of the application code. It may be very tricky and time consuming to locate and identify all of them unless the original developers of the application have used some technologies and software processes to enable quick search of any functionality implemented in the code. In terms of AOP, this task is referred to as aspect mining [29], extracting aspects from non-aspect-oriented programs. If there are no adequate tools and technologies to help software developers locate some functionality in existing applications, software engineers responsible for code maintenance have to explore the code “by hand” (or using simple text-searching tools such as grep) and often fail to perform the task completely (e.g., forget to find some of the scattered code fragments implementing the functionality). This causes more bugs and security flaws in the new version of the application. This is just one example of why AOP can be so helpful, since it supports aspect mining. In terms of AOP, the task of updating the scattered functionality can be considered as modifying the aspect and its join points within the target application. In AOP terms, adding functionality is considered as weaving the aspect that implements it.

In the 1980s, before the advent of AOP tools, my team used TIP technology [11], an enhancement of modular programming and abstract data types, to make the process of developing, self-documenting, and updating the code more systematic. Speaking in modern terms, TIP technology was based on a set of design and code templates using a predefined scheme of abstraction levels and vertical cuts (groups of operations). The application (e.g., a compiler) was designed as a related set of technological instrumental packages (TIPs), each responsible for implementing a set of abstract operations on a data structure (e.g., a table of definitions in the compiler). Each TIP was designed and implemented according to its predefined and recommended design template, typically (for compiler development projects) consisting of three abstract layers (in bottom-up order): representation level, definition level, and concept level; and four vertical cuts: creation/deletion interface, access interface, update interface, and output interface. Although for any other area (e.g., operating systems) the set of TIPs and their layers would be different, the principles of using the predefined “two-dimensional” horizontal layering and vertical operation grouping scheme would be the same. The users of TIPs (any other modules of the application) were granted access only to the TIP upper abstract (concept) layer, which supported adequate high-level abstract operations such as iterators and associative search. The code of each TIP was self-documented and self-explanatory, so that our colleagues from other institutions working on related compiler projects used the code of our TIP interfaces as working
documentation. Each operation of each TIP had a comfortable mnemonic name that helped quickly to locate all its uses by ordinary text-searching tools such as `grep`. This discipline of programming helped us not only to develop trustworthy and bug-free code, but also to update or enhance it quickly and trustworthy, since due to use of such technology, each code modification could be represented as a simple sequence of steps to update each abstract layer and each vertical cut. Much more detail on the TIP technology is provided in my book on Elbrus [11]. To summarize, our TIP technology was an important step leading to AOP and an understanding of its importance to trustworthy code development. Compared to AOP, what was not specified in a TIP (but what is specified in aspect specifications in modern AOP) is an explicit set of weaving rules on how to apply a TIP’s operations and where to insert its calls. Although we provided clear comments to the code, we didn’t use specific weaving tools and we inserted or modified the operations’ calls by hand. Nevertheless, our TIP technology experience was another school of trustworthy software development. Our latest experiences and results with AOP are described in this book.

1.3 ORGANIZATION OF THE BOOK

The structure of the book is very simple. The chapters are not strongly interdependent, so any chapter can be read or studied separately.

In this chapter, our primary goal has been to activate the readers’ interest and draw their attention to the problems considered in the book, to provide an initial understanding of them, and to illustrate them by examples and the results of my team’s long experience with software development.

In Chapter 2 we explain the history, essence, and modern aspects of trustworthy computing as a discipline. We see how TWC concepts originated, developed, and were brought to the attention of academia and industry by Microsoft. Microsoft’s TWC initiative and its four pillars—security, privacy, reliability, and business integrity—are explained and analyzed. Finally, we consider the implementation of TWC ideas and principles in the two latest software development platforms, Java and .NET.

Chapter 3 is devoted to aspect-oriented programming in whole and as related to our AOP framework for .NET, Aspect.NET in particular. We consider the history and basics of AOP, review both existing approaches to it and AOP tools, prevent the reader from being caught by some AOP “pitfalls,” and in the main focus of the chapter, consider in detail the principles of our approach to AOP and its implementation for .NET, our Aspect.NET framework, together with its advantages, features, use and perspectives.

Chapter 3 should be read before Chapter 4, since the latter describes principles and uses of AOP for trustworthy software development using Aspect.NET. A number of typical TWC-related tasks—such as synchronization, security checks, and design-by-contract checks—are considered in Chapter 4, and
recipes and examples are given on how to implement solutions to those tasks using AOP and our Aspect.NET framework. Following the traditions of the Wiley series of which this book is a part, a discussion of the use of quantitative analysis to improve productivity and efficiency is a primary focus. Applying AOP for TWC, we prove that solutions using Aspect.NET are as runtime efficient as those developed by hand, using performance tests. We provide a self-assessment of Aspect.NET using SQFD and ICED-T models. AOP and its roles are compared to some other popular approaches, such as agile software development. The main conclusion in Chapter 4 is that AOP with Aspect.NET is an adequate and efficient instrument for trustworthy software development. More examples of aspect definitions using Aspect.NET are provided in the Appendix.

Chapter 5 is for university teachers and students. In this chapter I summarize my teaching experience in the areas of TWC, AOP, and related domains. I describe my ERATO teaching paradigm, my principles of teaching TWC and AOP, and the contents of my courses and seminars related to them. Actually, all the university courses that I teach—secure software engineering, operating systems, compilers, .NET, and Java—appear to be closely related to TWC and penetrated by TWC ideas. My secure software engineering course contains a special chapter on AOP. All my courses are available at Microsoft Developer’s Network Academic Alliance Curriculum Repository (MSDNAA CR) Web site, so readers can download and use them for both teaching and self-training.

In Chapter 6 we outline perspectives on AOP and TWC: in particular, their relation to knowledge management, and some ideas on how to implement them.

The Appendix contains a self-documented code of aspect definitions to be used with Aspect.NET and the target applications to weave the aspects. The Appendix plays the role of practical addition to Chapter 4. The code of all samples is available at the Web site related to the book, www.aspectdotnet.org, and to the Aspect.NET project.
In this chapter we consider the concept of trustworthy computing (TWC) in detail: its historical roots, the growing need for TWC, its contemporary status and perspectives, and Microsoft’s TWC initiative and its four pillars: security, privacy, reliability, and business integrity. Two modern software development platforms, .NET and Java, are considered from a TWC viewpoint.

2.1 HISTORY OF AND GROWING NEED FOR TWC

As defined in a classic book on trustworthy computing [30]: “Trustworthiness is assurance that a system deserves to be trusted—that it will perform as expected despite environmental disruptions, human and operator error, hostile attacks, and design and implementation errors. Trustworthy systems reinforce the belief that they will continue to produce expected behavior and will not be susceptible to subversion.”

As emphasized by many authors, the concept of trustworthy computing is multidimensional and contains a lot of scientific, engineering, business, and
human factors. This book is devoted to only a part of this concept but one of the most important ones: trustworthy software development.

In my opinion, trustworthy computing issues arose with the appearance of the first computers, in the 1940s and early 1950s. Programs for those early computers and their data had to be input by the single user in person, using punched tapes, decks of punched cards, or an operator’s control panel. Any bug (e.g., hardware fault when reading the program or its data, an extra hole in a punched card, incorrectly set trigger on the control panel) could cause overall failure of program use, hanging up or halting the system without clear information regarding what happened. In addition, the use of explicit binary codes and concrete absolute addresses in programs made the process of coding and debugging critically unreliable. Early computers were untrustworthy because of their unreliable and poor user interface, ever when used by a single programmer to solve a single task. Their trustworthiness depended, first, on the trustworthiness, attention, and responsible behavior of the people who used them. Computer gurus of those times remembered by heart the right addresses to input programs into memory and the appropriate combinations of control panel triggers. The reason the computer stopped should have been guessed by solving the puzzle of LED combinations on the control panel. We’ll refer to all of these as primary TWC issues.

Some of the primary TWC issues were solved partially by using checksums of the input programs and data; others were resolved by using assembler and high-level languages. But software reliability TWC issues (e.g., buggy or malicious addressing of another task’s memory or accessing a resource already belonging to some other user or task) have been experienced up to now, even at a much higher level of application development than before.

Some papers on trustworthy computing say that trustworthy computing issues arose in the 1960s, when multiuser and multitasking computer systems appeared. For example [31], one of the first attempts to classify the goals and needs of trustworthy computing was undertaken in the mid-1960s, about 40 years before Microsoft’s TWC initiative, by the Allen-Babcock company, whose business was related to time-sharing computers. This is quite logical, since in any computer system consumed by several users and tasks simultaneously, common resource-sharing issues and related security, privacy, and reliability problems may arise at any moment, even without using LANs or the Internet. We’ll refer to the issues of TWC related to sharing computer resources by several users or tasks as sharing TWC issues.

For example, because of an unchecked bug in address arithmetic operation such as $p++$ in C, an application could easily intrude on some other application's memory area, or mistakenly access some other array of its own memory that differs from the one addressed by the $p$ pointer. In another example, because of a bug in the operating system, a race condition could occur for any types of resources common to different users or tasks: memory, hard disks, or CPU cycles.
Allen-Babcock’s approach to trustworthy computing [31] was based on the following principles, very similar to the four pillars of Microsoft’s trustworthy computing approach formulated in 2002 [32]:

- “An ironclad operating system [reliability]
- Use of trustworthy personnel [business integrity]
- Effective access control [security]
- User requested optional privacy [privacy]”

One of the major issues for most users in that “batch jobs and time-sharing era” was how to speed-up processing a job and thus maximize the number of turnarounds per day—starting with its input by the operator (or activating it from a terminal), until getting a listing with results. One of the widely used (but surely not correct) ways to do that was to negotiate with the operator, penetrate the computer room where the shared mainframe was working, use the operator’s console, increase the priority of your job while temporarily blocking others, and cut your listing from the printer and get away. From a contemporary TWC viewpoint, such types of behavior can be regarded as a “manual” attack that makes nontrustworthy use of the mainframe by others (regarded as your competitors), since they are unpleasantly surprised by what’s happening with their jobs and have to waste their working time. In the regular mode, all users shared the single mainframe computer (not networked to any others) and could, at most, start their jobs from their terminals, interact with them (input the job’s data and attach the debugger to the job), watch the remaining time the job was allowed to run in batch mode, and learn the job priority assigned by the OS according to the maximal time and memory resources claimed in the “passport” of the job at its start.

So in those batch times, one of the main security issues was to keep physical security in the computer room. Another was to keep the operator’s login and password away from users. In general, access control issues could arise when the user somehow phished another user’s or operator’s login and password and therefore could spoof the other user’s identity and consume or update resources or privileges to which he or she did not have access. Then came the era of networking, e-mails, the Internet, and the Web. Computing history says that regional networks such as ARPANET were created in the 1960s; e-mail came into wide use in the late 1970s; TCP/IP protocols, the basis of communications via the Internet, were developed in the 1970s; regional and global networks were integrated into the Internet in the late 1980s; HTML and the Web were invented in the early 1990s. In addition, the process of personalization of using computers (minicomputers in the 1970s and personal computers in the 1980s) connected into networks made possible many more TWC issues. Among them are spreading viruses, worms, and Trojan programs over the network; escalating hackers’ attacks, such as distributed denial of service (DDoS), to cause servers to crash; breaking the security of electronic control
systems of banks, military bases, and enterprises; corrupting Web sites of high importance; phishing users’ logins and passwords and account and credit cards numbers; and pharming users to malicious Web sites to steal their private information. Fifteen years ago one could hardly imagine that double-clicking an e-mail attachment to open it in your e-mail client application could be dangerous and might lead to infection of your computer and everything else in your LAN by an Internet worm. Dozens of spam e-mails intrude on each machine every day, despite spam filters, which sometimes filter out important e-mails from commercial companies or colleagues instead of spamlike advertisements. All of the issues described briefly here should be categorized as TWC networking issues.

In general, the primary, sharing, networking, and software reliability issues of trustworthy computing considered belong to major categories of TWC issues. Surprising as it may seem, all of them, except of course primary issues, have grown more and more, beginning from the early days of computing. The reason is as follows: The more complicated a system is, the more ways that can be used to break the system by an attack, and the higher the risk that it can be done. Hence, there is a growing need in systematic approaches to TWC, combining theory, methods, software development platforms, and tools. Although a lot of work has been done and a lot of progress has been achieved in the areas of TWC, these problems are still far from having a complete solution.

### 2.2 MICROSOFT’S TWC INITIATIVE

In January 2002, Bill Gates sent a historic e-mail [33] to all Microsoft employees, announcing the beginning of the trustworthy computing initiative. The essence of all aspects of the TWC initiative was explained in more detail in Craig Mundie’s white paper [32]. Another important foundation document on the TWC initiative is the second Bill Gates’ TWC e-mail [34], dated July 2002.

Although some authors [31] are still skeptical about the TWC initiative and regard it as a kind of justification of Microsoft’s security patches to Windows, Outlook, and other Microsoft’s products mostly subject to security attacks, I consider the TWC initiative from a more general viewpoint and think that this initiative plays a very important role, since it really stimulated many software engineering researchers, teachers, developers, and companies to more active joint efforts toward secure and reliable software. What is regrettable is that Microsoft didn’t announce the TWC initiative before—say, ten years ago. Otherwise, we’d now already have in use comfortable TWC frameworks and environments, and contemporary computing would already be much more trustworthy than it is now.

The four pillars of Microsoft’s trustworthy computing initiative are security, privacy, reliability, and business integrity. Their brief definitions, as the main
goals of TWC, in terms of software product customer’s expectations, are given by Mundie et al. [32]:

• **Security.** “The customer can expect that systems are resilient to attack, and that the confidentiality, integrity, and availability of the system and its data are protected.”

• **Privacy.** “The customer is able to control data about themselves, and those using such data adhere to fair information principles.”

• **Reliability.** “The customer can depend on the product to fulfill its functions when required to do so.”

• **Business integrity.** “The vendor of a product behaves in a responsive and responsible manner.”

Note that according to these definitions, the treatment of TWC by Microsoft is not limited to software and hardware issues. It is much wider: It includes business, legal, and social issues, emphasizes the role of computing in everyday life and work, and notes the issues to be resolved in this relation.

When participating in Microsoft academic days on trustworthy computing in April 2006 in Redmond [35], I appreciated the spirit and the principles of such a wide and general approach to TWC. As a software engineering expert, I was very interested to listen to the presentations on .NET security, Windows security, and writing secure code. But what pleasantly surprised me as an expert in knowledge management were the talks on intelligent software tools like those to help companies to keep their business legal, advising on the appropriate laws from the knowledge base. I agree with Microsoft that such tools are also part of contemporary TWC, in addition to principles of trustworthy software development.

It should also be emphasized that despite skeptics’ opinions [31], Microsoft has, due to its TWC initiative, not only promptly reorganized its software product development and testing business to make the products more trustworthy, but is also demonstrating a positive example and appealing to other software companies to act similarly and participate in the TWC initiative. The reason that Microsoft’s software products (and Microsoft itself) are so subject to attack is clear: Microsoft’s operating systems, office applications, and software development tools have been among the most widely used all around the world for over 30 years.

Of the four TWC pillars formulated by Microsoft, security and reliability are the most traditional. For many people, the motto of TWC is associated first with security: encrypting programs and data, struggling with viruses and worms, mitigating network attacks, and so on. **Reliability** is an equally important area for TWC, since nonreliable software is vulnerable to attack based on its known bugs and other problems, and may cause insecure and unsafe situations.

The remaining two TWC pillars, **privacy** and **business integrity**, are dependent primarily on people’s activity and behavior, since they not only require people to keep the software business correct and legal (as to private and
confidential information), but must also recommend following software process
disciplines aimed at making software products trustworthy. So the human
factors remain an important part of TWC. What software technologies can do
to make the human factors more trustworthy is to implement and offer appro-
priate suggestions and software process disciplines to users.

Implementation of intelligent advice related to changing situation, gener-
ally speaking, requires knowledge management technologies and tools that use
an extensible domain-specific knowledge base. So I think, in the near perspec-
tive, that knowledge management should become an inherent part of TWC
tools. As one present opportunity to represent and use knowledge for TWC
purposes and to implement intelligent TWC solutions, software developers
working for the Microsoft.NET platform can use our knowledge management
tool, Knowledge.NET [36], which makes it possible to combine knowledge
management and traditional .NET software engineering techniques and
features, using our extension of C# by ontologies, frames, and rule sets
while working in a Visual Studio.NET development environment, with
Knowledge.NET integrated as an add-in. Detailed consideration of
Knowledge.NET is beyond the scope of this book.

In my opinion, all four pillars of TWC are very important, but I think a fifth
one equally important should be added: usability—ease of use and a user-
friendly interface. When assessing whether or not a product is trustworthy, this
quality of any software product is so important from the user’s viewpoint that
he or she will not use the product if it is not attractive for some reason and
its appearance and operating interface are awkward and uncomfortable. I
noted earlier the nontrustworthy interface of early computers, based on trig-
gers and LEDs on the control panel. That “growth shortcoming” was overcome
long ago. But from time to time, software products appear that force users to
perform a number of redundant actions. They impose on users some nonfl ex-
ible disciplines of operation which most users do not agree with but are unable
to customize. Their output consists of incomplete, unclear, or silly messages.
Therefore, users waste their time and, as a result, are pushed away not only
from using these products, but also from using computing technologies, for a
long time. No usability—no trust; so I think that usability features are as
important as security, reliability, and privacy from a TWC viewpoint. In the
TWC white paper [32], usability is defined as one means to achieve the four
main goals, but in my opinion, it should be one of the TWC pillars, as it is a
crucial for product quality.

2.3 THE FOUR PILLARS OF TWC

2.3.1 Security

Definitions of security differ, both in the general meaning of the word and as
related to computers. Here are two of them which I think explain best the
essence of this very important concept:
• “Security is the condition of being protected against danger or loss … that originate[s] from outside” [37].
• As applicable to computing, “computer security is a field of computer science concerned with the control of risks related to computer use” [38].

There are three parties to the security paradigm: the system to be secured; the user, who is eager that the system be secure; and the attackers, who are trying to break into the system. The most important focus in any definition of computer security is on explicit systematic measures to be taken to keep a system secure. These security measures are aimed, first, at guarding the system from attack, and second, at keeping its functionality comfortable for use, despite security actions implemented in the product. The user of the system should understand the alternatives and grades regarding how he or she can protect the system, what types of security checks will be performed at each security level, and how his or her activity will or should reasonably be restricted at each security level to achieve the desired level of security.

The following major tasks can be formulated in relation to computer security:

1. Classification and analysis of systems and their types of security, categories of users (from a security and allowable actions viewpoint), known and imaginable types of attacks and attackers, and possible security measures to be taken. There is a lot of very interesting literature in this area: in particular, the book by Howard and LeBlanc [27], the best one I know.
2. Finding ways to use quantitative assessment of security. This task is one of the most complicated—an intellectual challenge to be solved. As far as I can judge, we are now only starting on our path to solving it. Using mathematical formulas in terms of probability theory and statistics would be too simplified an approach, due to the polymorphic and complicated nature of security and a lot of human factors that participate. But clearly, any scientific approach to such a complicated problem as computer security should include its quantitative evaluation.
3. More practical and traditional for software engineering up to now has been developing security technologies and tools. A lot of progress has been made in this direction. Actually, each operating system and software development platform pays a lot of attention to the security technologies involved, and this tendency is growing stronger.

This book deals primarily with solving the third task, although in Chapter 4 we touch on the second.

In a more practical and ubiquitous sense, for home users, security involves primarily everyday struggles with viruses and worms: from flash memory