1

Introduction

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

Using Aspect-Oriented Programming for Trustworthy Software Development,
By Vladimir O. Safonov
Copyright © 2008 John Wiley & Sons, Inc.

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 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 trustworthy 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 checking, 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 intermediate 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 trustworthily, 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.