This book is about aspect-oriented software development (AOSD), a set of emerging technologies that seeks new modularizations of software systems. AOSD allows multiple concerns to be separately expressed but nevertheless be automatically unified into working systems. We intend this book to be an overview of these technologies for the computer professional interested in learning about state-of-the-art developments.

In general, programming is about realizing a set of requirements in an operational software system. One has a (perhaps evolving) set of properties desired of a system, and proceeds to develop that system to achieve those properties. Software engineering is the accumulated set of processes, methodologies, and tools to ease that evolutionary process, including techniques for figuring out what we want to build and mechanisms for yielding a higher-quality resulting system.

A recurrent theme of software engineering (and engineering in general) is that of modularization: “separation and localization of concerns.” That is, we have “concerns”—things we care about—in engineering any system. These concerns range from high-level, user-visible requirements like reliability and security to low-level implementation issues like caching and synchronization. Ideally, separating concerns in engineering simplifies system development by allowing the development of specialized expertise and by producing an overall more comprehensible arrangement of elements.

Traditional software development has focused on decomposing systems into units of primary functionality, recognizing the other issues of concern, and leaving it to programmers both to code modules corresponding to the primary functionality and to make sure that all other issues of concern are addressed in the code wherever appropriate. Sometimes these other concerns can be packaged into modules of
behavior themselves (e.g., subroutines, methods, or procedures). However, often the degree of shared context or the cost of contextual change (for example, the cost of a subprogram call) necessitates intermixing (crosscutting) the instructions for the primary functionality and the other concerns. In any case, conventional development requires programmers to keep in mind all the things that need to be done, how to deal with each issue, the problems associated with the possible interactions, and the execution of the right behavior at the right time.

Spreading out the responsibility for invoking the code for multiple concerns to all programmers produces a more brittle system. Each programmer who has to do something right is one more person who can make a mistake; each spot where something needs to be done is a potential maintenance mishap. The distribution of the code for realizing a concern becomes especially critical as the requirements for that concern evolve—a system maintainer must find and correctly update a variety of (likely poorly identified) situations.

We use the term aspect-oriented programming (AOP) to describe the activity of programming with multiple crosscutting concerns or aspects. The general modus operandi of programming AOSD systems is to let system developers express the behavior for each concern in its own module. Such a system must also include some directions for how the different concerns are to be knitted together into a working system (for example, to which program entities each separate concern applies) and a mechanism for actually producing a working system from these elements. For example, many AOSD systems provide a way to say, “High security is achieved by doing X. Reliability is achieved by doing Y. I want high security in the following places in the code and reliability on these operations.” The AOSD system then produces an object that invokes the high security and reliability codes appropriately.

Concern-level illustrations of the application of AOP techniques include replication, configuration, debugging, mobility, program instrumentation, security, code movement, and synchronization. AOP is only beginning to penetrate commercial applications, but interesting prototypes have been demonstrated in areas such as application servers, operating systems kernels, real-time distributed event channels, distributed middleware, distributed quality-of-service, multi-agent system architectures, object databases, domain-specific visual modeling, collaboration, workflow, e-commerce, software visualization, engineering design, and data processing.

1.1 BOOK ORGANIZATION

Just like object-oriented programming, aspect technology started with aspect-oriented programming languages. Currently, several aspect-oriented languages are in widespread availability, and researchers are continually inventing new ones. Part 1
of this book, “Languages and Foundations,” examines this area, including descriptions of not only proposed AOP languages but also programming models based on aspect ideas and chapters discussing the fundamental and historical nature of AOP.

Just as object-oriented programming led to the development of a large class of object-oriented development methodologies, AOP has encouraged a nascent set of software engineering technologies. Part 2, “Software Engineering,” examines these issues, include methodologies for dealing with aspects, modeling techniques (often based on the ideas of the Unified Modeling Language, UML), and testing technology for assessing the effectiveness of aspect approaches.

Of course, the ultimate aim of programming is to develop software systems. In Part 3, “Applications,” we present descriptions of the application of aspect technology to particular software problems, including examples that range from the systems to application levels.

Each part of the book includes an introduction to that area and chapters by contributors describing their own work. We invited each contributor to either create an original chapter, targeted at the advanced programmer, or to nominate reprinting an existing paper meeting those criteria. Several of our contributors chose to meld the two approaches, revising existing work in light of new experience and the intended audience.

AOSD is a rapidly evolving area. This format has enabled us to present the reader with a wider overview, a more current set of work, and a clearer sense of the diversity of opinions than a synopsis of different research or an in-depth study of one particular research direction would have provided.

### 1.2 Common Terminology

Certain common terminology and themes pervade these papers. Different authors have variations on the meaning they assign to these ideas. This is a symptom of intellectual youth and ferment. They use certain terms freely. Thus, it is helpful to begin with a brief glossary of common AOSD concepts.

**Concerns.** Any engineering process has many things about which it cares. These range from high-level requirements (“The system shall be manageable”) to low-level implementation issues (“Remote values shall be cached”). Some concerns are localized to a particular place in the emerging system (“When the M key is pressed, the following menu shall pop up”), some refer to measurable properties of the system as a whole (“Response time shall be less than a second”), others are aesthetic (“Programmers shall use meaningful variable names”), and others involve systematic behavior (“All database changes shall be logged”). Generically,
we call all these concerns, though AOSD technology is particularly directed at the last, systematic class.

**Crosscutting concerns.** Software development addresses concerns, both concerns at the user/requirements levels and at the design/implementation level. Often, the implementation of one concern must be scattered throughout the rest of an implementation. We say that such a concern is *crosscutting*. Note that what is crosscutting is a function of both the particular decomposition of a system and the underlying support environment. A particular concern might crosscut in one view of an architecture while being localized in another; a particular environment might invisibly support a concern (for example, security) that needs to be explicitly addressed in another.

**Code tangling.** In conventional environments, implementing crosscutting concerns usually results in code tangling—the code for concerns becomes intermixed. Ideally, software engineering principles instruct us to modularize our system software in such a way that (1) each module is cohesive in terms of the concerns it implements and (2) interfaces between modules are simple. Software that complies with these principles tends to be easier to produce, more naturally distributed among different programmers, easier to verify and test, and easier to maintain, reuse, and evolve to future requirements. Crosscutting works against modularization. Code for crosscutting concerns finds itself scattered through multiple modules; changes to that code now require changing all the places it touches, and (perhaps more importantly and less obviously) all changes to the system must conform to the requirements of the crosscutting concern. That is, if certain actions require, say, a security or accounting action, then in maintaining the code, we must consider how every change interacts with security and accounting.

**Aspects.** An *aspect* is a modular unit designed to implement a concern. An aspect definition may contain some code (or *advice*, which follows) and the instructions on where, when, and how to invoke it. Depending on the aspect language, aspects can be constructed hierarchically, and the language may provide separate mechanisms for defining an aspect and specifying its interaction with an underlying system.

**Join points.** *Join points* are well-defined places in the structure or execution flow of a program where additional behavior can be attached. A join point model (the kinds of joint points allowed) provides the common frame of reference to enable the definition of the structure of aspects. The most common elements of a join point model are method calls, though aspect languages have also defined join points for a variety of other circumstances, including field definition, access, and modification, exceptions, and execution events and states. For example, if an AOP
language has method calls in its join point model, a programmer may designate additional code to be run on particular method calls.

**Advice.** Advice is the behavior to execute at a join point. For example, this might be the security code to do authentication and access control. Many aspect languages provide mechanisms to run advice before, after, instead of, or around join points of interest. Advice is oblivious in that there is no explicit notation at the joint point that the advice is to be run here—the programmer of the original base code may be oblivious to the evolving requirements. This contrasts with conventional programming languages, where the most common concern modularization mechanism, the subprogram, must be explicitly called.

**Pointcut designator.** A pointcut designator describes a set of join points. This is an important feature of AOP because it provides a quantification mechanism—a way to talk about doing something at many places in a program with a single statement. A programmer may designate all the join points in a program where, for example, a security code should be invoked. This eliminates the need to refer to each join point explicitly and thereby reduces the likelihood that any aspect code would be incorrectly invoked.

**Composition.** Abstractly, the idea of bringing together separately created software elements is composition. Different languages provide a variety of composition techniques, including subprogram invocation, inheritance, and generic instantiation. An important software engineering issue in the composition of components is the guarantees and mechanisms that a language provides to make sure that elements being composed “fit together.” This allows warning of incompatibilities during system development, rather than being surprised by them during system execution. Common mechanisms for such guarantees include type checking the signatures of subprogram calls and the interface mechanism of languages like Java.

**Weaving.** Weaving is the process of composing core functionality modules with aspects, thereby yielding a working system. Various AOP languages have defined several mechanisms for weaving, including statically compiling the advice together with base code, dynamically inserting aspects when loading code, and modifying the system interpreter to execute aspects.

**Wrapping, before and after.** One of the most common AOP techniques is to provide method calls as (sometimes the only) join points and to allow the advice to run either before, after, or around the method call. This notion can be generalized into the idea of wrapping—providing a filter or container around a component, which mediates communications to that component and enforces the desired aspects.
Statics and dynamics. The terms static and dynamic appear in some of the following discussions. In general, static elements are ones that can be determined before the program begins execution, typically at compile time; dynamic things happen at execution. A weaving process can be either static or dynamic, depending on whether it relies on a compilation or loading mechanism (static) or run-time monitoring (dynamic) for its realization. Somewhat orthogonally, an AOP language can be characterized as having static or dynamic join points, depending on whether the places that aspects are to be invoked are dependent purely on the compile-time structure of the original code or the run-time events of program execution.

The tyranny of the dominant decomposition. The previous discussion spoke of aspects as something to be imposed on a “base” program. However, one can make a perfectly coherent argument that all code elements should be treated as equals and that the best way to build AOSD systems is by providing a language for weaving together such elements. Some of the systems discussed in this book adopt that point of view. A key subtext of that discussion is whether aspect behavior can be imposed on aspects themselves. Doing so makes the contractual assertions of aspects more complete at the cost of complicating the underlying implementations.

1.3 HISTORICAL CONTEXT

The history of programming has been a slow and steady climb from the depths of direct manipulation of the underlying machines to linguistic structures for expressing higher-level abstractions. Progress in programming languages and design methods has always been driven by the invention of structures that provide additional modularity. Subroutines assembled the behavior of unstructured machine instructions, structured programming argued for semantic meaning for these subroutines, abstract data types recognized the unity of data and behavior, and object-orientation (OO) generalized this to multiplicity of related data and behaviors.

The current state-of-the-art in programming is object-oriented (OO) technology. With objects, the programmer is supposed to think of the universe as a set of instances of particular classes that provide methods, expressed as imperative programs, to describe the behavior of all the objects of a class.

Object-orientation has many virtues, particularly in comparison to its predecessors. Objects provide modularization. The notion of sending messages to objects helps concentrate the programmer’s thinking and aids understanding code. Inheritance mechanisms in object systems provide a way both to ascribe related behaviors to multiple classes and to make exceptions to that prescription.
Objects are not the last word in programming organization. This book is about an emerging candidate for the next step in this progression, aspect-oriented software development. Aspects introduce new linguistic mechanisms to modularize the implementation of concerns. Each of the earlier steps (with the minor exception of multiple inheritance in OO systems) focused on centralizing on a primary concern. AO, like its predecessors, is about recognizing that software systems are built with respect to many concerns and that programming languages, environments, and methodologies must support modularization mechanisms that honor these concurrent concerns. AO is technology for extending the kinds of concerns that can be separately and efficiently modularized.