CHAPTER 1

Object-Oriented Frameworks for Network Programming

CHAPTER SYNOPSIS

Object-oriented frameworks help reduce the cost and improve the quality of networked applications by reifying software designs and pattern languages that have proven effective in particular application domains. This chapter illustrates what frameworks are and compares them with other popular software development techniques, such as class libraries, components, patterns, and model-integrated computing. It then illustrates the process of applying frameworks to networked applications and outlines the ACE frameworks that are the focus of this book. These frameworks are based on a pattern language [POSA1, POSA2] that has been applied to thousands of production networked applications and middleware worldwide.

1.1 An Overview of Object-Oriented Frameworks

Even as computing power and network bandwidth increase dramatically, the development of networked application software remains expensive, time consuming, and error prone. The cost and effort stems from the growing demands placed on networked software, as well as the continual rediscovery and reinvention of core software design and implementation artifacts throughout the software industry. Moreover, the heterogeneity of hardware architectures, diversity of OS and network platforms, and stiff global competition makes it increasingly hard to build high-quality networked application software from scratch.

The key to building high-quality networked software in a time-to-market-driven environment is the ability to reuse successful software designs and implementations that have already been developed. Reuse has been a popular topic of debate and discussion for over 30 years in the software community [McI68]. There are two general types of reuse:

- **Opportunistic reuse,** in which developers cut and paste code from existing programs to create new ones. Opportunistic reuse works in a limited way for individual programmers or small groups. It doesn't scale up across business units or enterprises, however, and therefore doesn't significantly reduce development cycle time and cost or improve software quality. Worse, opportunistic reuse can actually impede development progress since cut-and-paste code often begins to diverge as it proliferates, forcing developers to fix the same bugs multiple times in multiple places.
- **Systematic reuse,** which is an intentional and concerted effort to create and apply multiuse software architectures, patterns, frameworks, and components throughout a product line [CN02]. In a well-honed systematic reuse process, each new project leverages time-proven designs and implementations, only adding new code that's specific to a particular application. This type of reuse is essential to increase software productivity and quality by breaking the costly cycle of rediscovering, reinventing, and revalidating common software artifacts.

Middleware [SS02] is a class of software that can increase systematic reuse levels significantly by functionally bridging the gap between the end-to-end functional requirements of networked applications and the underlying operating systems and network protocol stacks. Middleware provides capabilities that are critical to networked applications because they automate common network programming tasks. Developers who use middleware can therefore program their networked applications more like stand-alone applications, rather than wrestling with the many tedious and error-prone details associated with low-level OS event demultiplexing, message buffering and queueing, marshaling and demarshaling, and connection management mechanisms. Popular examples of middleware include Java virtual machines (JVMs), Enterprise JavaBeans (EJB), .NET, the Common Object Request Broker Architecture (CORBA), and the ADAPTIVE Communication Environment (ACE).

Systematically developing high-quality, reusable middleware for networked applications presents many hard technical challenges, including

- Detecting and recovering from transient and partial failures of networks and hosts in an application-independent manner
- Minimizing the impact of latency and jitter on end-to-end application performance
- Determining *how* to partition a distributed application into separate component services
- Deciding *where* and *when* to distribute and load balance services in a network

Since reusable middleware is inherently abstract, it's hard to validate its quality and to manage its production. Moreover, the skills required to develop, deploy, and support reusable networked application middleware have traditionally been a "black art," locked in the heads of expert developers and architects. These technical impediments to systematic reuse are often exacerbated by a myriad of nontechnical impediments [Hol97], such as organizational,

economic, administrative, political, sociological, and psychological factors. It's therefore not surprising that significant levels of software reuse have been slow to materialize in many projects and organizations [Sch00].

While it's never easy to make reuse work universally, we've led the development of powerful *host infrastructure middleware* called ACE that's designed specifically with systematic reuse in mind. During the past decade, we've written hundreds of thousands of lines of C++ code while developing and applying ACE to networked applications as part of our work with dozens of telecommunication, aerospace, medical, and financial services companies. As a result of our experience, we've documented many patterns and pattern languages [POSA2, POS00] that have guided the design of reuseable middleware and applications. In addition, we've taught hundreds of tutorials and courses on reuse, middleware, and patterns to thousands of developers and students. Despite the many technical and nontechnical challenges, we've identified a solid body of work that combines advanced research, time-proven design knowledge, hands-on experience, and software artifacts that can significantly enhance the systematic reuse of networked application software.

At the heart of this body of work are object-oriented frameworks [FJS99b, FJS99a], which are a powerful technology for achieving systematic reuse of networked application software.[1] Below, we describe the three characteristics of frameworks [JF88] that help them to achieve the important networked application qualities listed on page xi. Figure 1.1 (page 4) illustrates how these characteristics work together.

A framework provides an integrated set of domain-specific structures and functionality. Systematic reuse of software depends largely on how well frameworks model the commonalities and variabilities [CHW98] in application domains, such as business data processing, telecom call processing, graphical user interfaces, or distributed object computing middleware. Since frameworks reify the key roles and relationships of classes in application domains, the amount of reusable code increases and the amount of code rewritten for each application decreases.

A framework exhibits "inversion of control" at run time via callbacks. A *callback* is an object registered with a dispatcher that calls back to a method on the object when a particular event occurs, such as a connection request or data arriving on a socket handle. Inversion of control decouples the canonical detection, demultiplexing, and dispatching steps within a framework from the application-defined *event handlers* managed by the framework. When events occur, the framework calls back to virtual *hook methods* in the registered event handlers, which then perform application-defined processing in response to the events.

Since frameworks exhibit inversion of control, they can simplify application design because the framework—rather than the application—runs the event loop to detect events, demultiplex events to event handlers, and dispatch hook methods on the handlers that process

[1] In the remainder of this book we use the term *framework* to mean *object-oriented framework*.

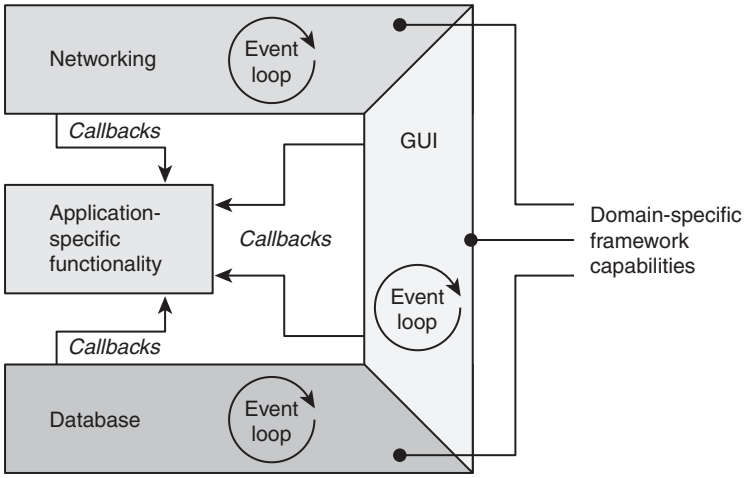

Figure 1.1: Synergy of Framework Capabilities

the events. The use of virtual hook methods in the handler classes decouples the application's classes from the framework, allowing each to be changed independently as long as the interface signature and interaction protocols aren't modified.

A framework is a "semi-complete" application that programmers can customize to form complete applications by inheriting from and instantiating classes in the framework. Inheritance enables the features of framework base classes to be shared selectively by subclasses. If a base class provides default implementations of its methods, application developers need only override those virtual methods whose default behavior doesn't meet their needs.

Since a framework is a semi-complete application, it enables larger-scale reuse of software than can be achieved by reusing individual classes or stand-alone functions. The amount of reuse increases due to a framework's ability to integrate application-defined and application-independent classes. In particular, a framework abstracts the canonical control flow of applications in a domain into families of related classes, which can collaborate to integrate customizable application-independent code with customized application-defined code.

1.2 Comparing Software Development and Reuse Techniques

Object-oriented frameworks don't exist in isolation. Class libraries, components, patterns, and model-integrated computing are other techniques that are being applied to reuse software and increase productivity. This section compares frameworks with these techniques to illustrate their similarities and differences, as well as to show how the techniques can be combined to enhance systematic reuse for networked applications.

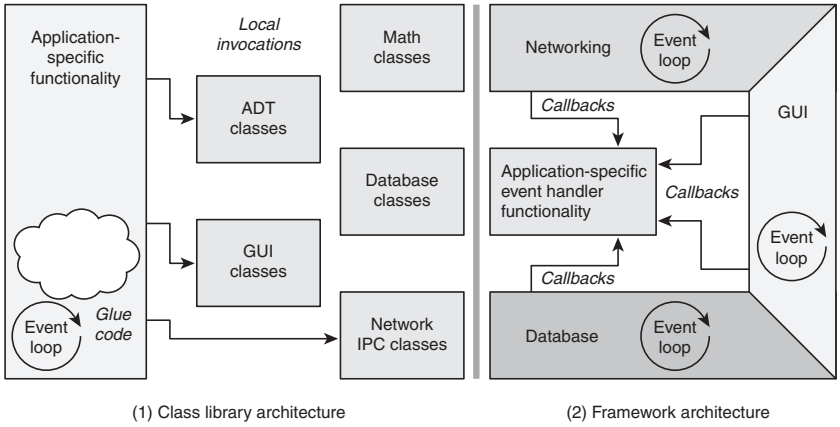

Figure 1.2: Class Library versus Framework Architectures

1.2.1 Comparing Frameworks and Class Libraries

A class is a general-purpose, reusable building block that specifies an interface and encapsulates the representation of its internal data and the functionality of its instances. A library of classes was the most common first-generation object-oriented development technique [Mey97]. Class libraries generally support reuse-in-the-small more effectively than function libraries since classes emphasize the cohesion of data and methods that operate on the data.

Although class libraries are often domain independent and can be applied widely, their effective scope of reuse is limited because they don't capture the canonical control flow, collaboration, and variability among families of related software artifacts. The total amount of reuse with class libraries is therefore relatively small, compared with the amount of application-defined code that must be rewritten for each application. The need to reinvent and reimplement the overall software architecture and much of the control logic for each new application is a prime source of cost and delay for many software projects.

The C++ standard library [Bja00] is a good case in point. It provides classes for strings, vectors, and other containers. Although these classes can be reused in many application domains, they are relatively low level. Application developers are therefore responsible for (re)writing much of the "glue code" that performs the bulk of the application control flow and class integration logic, as shown in Figure 1.2 (1).

Frameworks are a second-generation development technique [Joh97] that extends the benefits of class libraries in several ways. Most importantly, classes in a framework collaborate to provide a reusable architecture for a family of related applications. Class collaboration in a framework yields "semi-complete" applications that embody domain-specific object structures and functionality. Frameworks can be classified by various means, such as the blackbox and whitebox distinctions described in Sidebar 1 (page 6).

Sidebar 1: Overview of Whitebox and Blackbox Frameworks

Frameworks can be classified in terms of the techniques used to extend them, which range along a continuum from *whitebox frameworks* to *blackbox frameworks* [HJE95], as described below:

- **Whitebox frameworks.** Extensibility is achieved in a whitebox framework via object-oriented language features, such as inheritance and dynamic binding. Existing functionality can be reused and customized by inheriting from framework base classes and overriding predefined hook methods [Pre95] using patterns such as Template Method [GoF], which defines an algorithm with some steps supplied by a derived class. To extend a whitebox framework, application developers must have some knowledge of its internal structure.
- **Blackbox frameworks.** Extensibility is achieved in a blackbox framework by defining interfaces that allow objects to be plugged into the framework via composition and delegation. Existing functionality can be reused by defining classes that conform to a particular interface and then integrating these classes into the framework using patterns such as *Function Object* [Kuh97], *Bridge*/*Strategy* [GoF], and *Pluggable Factory* [Vli98b, Vli99, Cul99], which provide a blackbox abstraction for selecting one of many implementations. Blackbox frameworks can be easier to use than whitebox frameworks since application developers need less knowledge of the framework's internal structure. Blackbox frameworks can also be harder to design, however, since framework developers must define crisp interfaces that anticipate a range of use cases.

Another way that class libraries differ from frameworks is that the classes in a library are typically passive since they perform their processing by borrowing the thread from so-called self-directed applications that invoke their methods. As a result, developers must continually rewrite much of the control logic needed to bind the reusable classes together to form complete networked applications. In contrast, frameworks are active since they direct the flow of control within an application via various callback-driven event handling patterns, such as Reactor [POSA2] and Observer [GoF]. These patterns invert the application's flow of control using the *Hollywood Principle*: "Don't call us, we'll call you" [Vli98a]. Since frameworks are active and manage the application's control flow, they can perform a broader range of activities on behalf of applications than is possible with passive class libraries.

Frameworks and class libraries are complementary technologies in practice. Frameworks provide a foundational structure to applications. Since frameworks are focused on a specific domain, however, they aren't expected to satisfy the broadest range of application development needs. Class libraries are therefore often used in conjunction within frameworks and applications to implement commonly needed code artifacts, such as strings, files, and time/date classes.

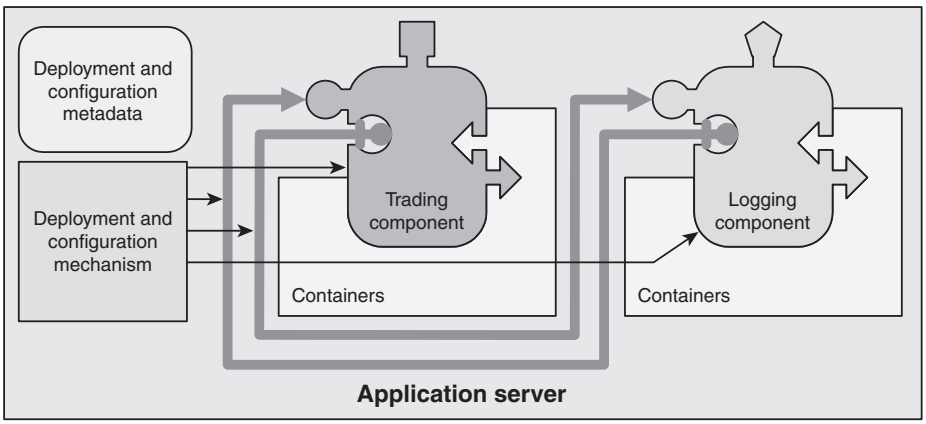

Figure 1.3: A Component Architecture

For example, the ACE frameworks use the ACE wrapper facade classes to ensure their portability. Likewise, applications can use the ACE container classes described in [HJS] to help implement their event handlers. Whereas the ACE container classes and wrapper facades are passive, the ACE frameworks are active and provide inversion of control at run time. The ACE toolkit provides both frameworks and a library of classes to help programmers address a range of challenges that arise when developing networked applications.

1.2.2 Comparing Frameworks and Components

A component is an encapsulated part of a software system that implements a specific service or set of services. A component has one or more interfaces that provide access to its services. Components serve as building blocks for the structure of an application and can be reused based solely upon knowledge of their interface protocols.

Components are a third-generation development technique [Szy98] that are widely used by developers of multitier enterprise applications. Common examples of components include ActiveX controls [Egr98] and COM objects [Box98], .NET web services [TL01], Enterprise JavaBeans [MH01], and the CORBA Component Model (CCM) [Obj01a]. Components can be plugged together or scripted to form complete applications, as shown in Figure 1.3.

Figure 1.3 also shows how a component implements the business application logic in the context of a container. A container allows its component to access resources and services provided by an underlying middleware platform. In addition, this figure shows how generic application servers can be used to instantiate and manage containers and execute the components configured into them. Metadata associated with components provide instructions that application servers use to configure and connect components.

Many interdependent components in enterprise applications can reside in multiple—possibly distributed—application servers. Each application server consists of some number of components that implement certain services for clients. These components in turn may include other *collocated* or remote services. In general, components help developers reduce their initial software development effort by integrating custom application components with reusable off-the-shelf components into generic application server frameworks. Moreover, as the requirements of applications change, components can help make it easier to migrate and redistribute certain services to adapt to new environments, while preserving key application properties, such as security and availability.

Components are generally less lexically and spatially coupled than frameworks. For example, applications can reuse components without having to subclass them from existing base classes. In addition, by applying common patterns, such as Proxy [GoF] and Broker [POSA1], components can be distributed to servers throughout a network and accessed by clients remotely. Modern application servers, such as JBoss and BEA Systems's WebLogic Server, use these types of patterns to facilitate an application's use of components.

The relationship between frameworks and components is highly synergistic, with neither subordinate to the other [Joh97]. For example, the ACE frameworks can be used to develop higher-level application components, whose interfaces then provide a facade [GoF] for the internal class structure of the frameworks. Likewise, components can be used as pluggable strategies in blackbox frameworks [HJE95]. Frameworks are often used to simplify the development of middleware component models [TL01, MH01, Obj01a], whereas components are often used to simplify the development and configuration of networked application software.

1.2.3 Comparing Frameworks and Patterns

Developers of networked applications must address design challenges related to complex topics, such as connection management, service initialization, distribution, concurrency control, flow control, error handling, event loop integration, and dependability. Since these challenges are often independent of specific application requirements, developers can resolve them by applying the following types of patterns [POSA1]:

- **Design patterns** provide a scheme for refining the elements of a software system and the relationships between them, and describe a common structure of communicating elements that solves a general design problem within a particular context.
- **Architectural patterns** express the fundamental, overall structural organization of software systems and provide a set of predefined subsystems, specify their responsibilities, and include guidelines for organizing the relationships between them.
- **Pattern languages** define a vocabulary for talking about software development problems and provide a process for the orderly resolution of these problems.

Traditionally, patterns and pattern languages have been locked in the heads of expert developers or buried deep within the source code of software applications and systems. Allowing this valuable information to reside only in these locations is risky and expensive. Explicitly capturing and documenting patterns for networked applications helps to

- **Preserve important design information** for programmers who enhance and maintain existing software. This information will be lost if it isn't documented, which can increase software entropy and decrease software maintainability and quality.
- **Guide design choices** for developers who are building new applications. Since patterns document the common traps and pitfalls in their domain, they help developers to select suitable architectures, protocols, algorithms, and platform features without wasting time and effort (re)implementing solutions that are known to be inefficient or error prone.

Knowledge of patterns and pattern languages helps to reduce development effort and maintenance costs. Reuse of patterns alone, however, does not create flexible and efficient software. Although patterns enable reuse of abstract design and architecture knowledge, software abstractions documented as patterns don't directly yield reusable code. It's therefore essential to augment the study of patterns with the creation and use of frameworks. Frameworks help developers avoid costly reinvention of standard software artifacts by reifying common patterns and pattern languages and by refactoring common implementation roles.

ACE users can write networked applications quickly because the frameworks in ACE implement the core patterns associated with service access, event handling, concurrency, and synchronization [POSA2]. This knowledge transfer makes ACE more accessible and directly applicable compared to many other common knowledge transfer activities, such as seminars, conferences, or design and code reviews. Although these other activities are useful, they are limited because participants must learn from past work of others, and then try to apply it to their current and future projects. In comparison, ACE provides direct knowledge transfer by embodying framework usage patterns in a powerful toolkit containing both networked application domain experience *and* working code.

For example, JAWS [HS99] is a high-performance, open-source, adaptive Web server built using the ACE frameworks. Figure 1.4 (page 10) illustrates how the JAWS Web server is structured as a set of collaborating frameworks whose design is guided by the patterns listed along the borders of the figure. These patterns help resolve common design challenges that arise when developing concurrent servers, including encapsulating low-level operating system APIs, decoupling event demultiplexing and connection management from protocol processing, scaling up server performance via multithreading, minimizing server threading overhead, using asynchronous I/O effectively, and enhancing server configurability. More information on the patterns and design of JAWS appears in Chapter 1 of POSA2.

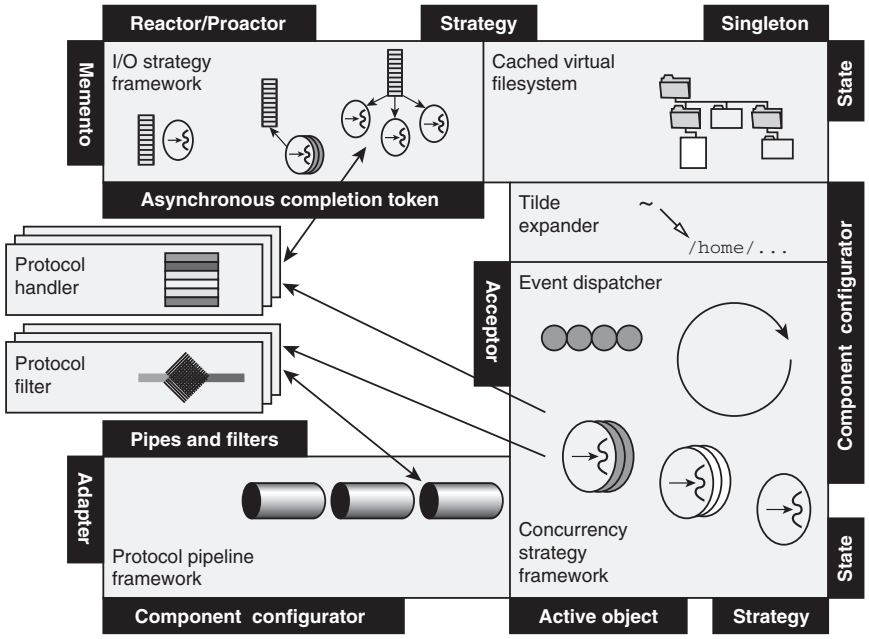

Figure 1.4: Patterns Forming the Architecture of JAWS

1.2.4 Comparing Frameworks and Model-Integrated Computing

Model-integrated computing (MIC) [SK97] is an emerging development paradigm that uses domain-specific modeling languages to systematically engineer software ranging from small-scale real-time embedded systems to large-scale enterprise applications. MIC development environments include domain-specific model analysis and model-based program synthesis tools. MIC models can capture the essence of a class of applications, as well as focus on a single, custom application. MIC also allows the modeling languages and environments themselves to be modeled by so-called *meta-models* [SKLN01], which help to synthesize domain-specific modeling languages that can capture subtle insights about the domains they are designed to model, making this knowledge available for reuse.

Popular examples of MIC being used today include the Generic Modeling Environment (GME) [LBM+01] and Ptolemy [BHLM94] (which are used primarily in the real-time and embedded domain) and UML/XML tools based on the OMG Model Driven Architecture (MDA) [Obj01b] (which are used primarily in the business domain thus far). When implemented properly, these MIC technologies help to

- Free application developers from dependencies on particular software APIs, which ensures that the models can be reused for a long time, even as existing software APIs are obsoleted by newer ones.

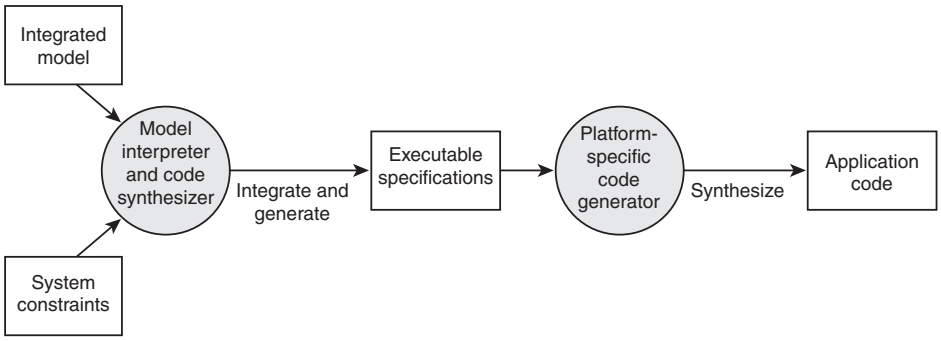

Figure 1.5: Steps in the Model-Integrated Computing Development Process

- Provide correctness proofs for various algorithms by analyzing the models automatically and offering refinements to satisfy various constraints.
- Generate code that's highly dependable and robust since the modeling tools themselves can be synthesized from meta-models using provably correct technologies.
- Rapidly prototype new concepts and applications that can be modeled quickly using this paradigm, compared to the effort required to prototype them manually.
- Reuse domain-specific modeling insights, saving significant amounts of time and effort, while also reducing application time-to-market and improving consistency and quality.

As shown in Figure 1.5, the MIC development process uses a set of tools to analyze the interdependent features of the application captured in a model and determine the feasibility of supporting different QoS requirements in the context of the specified constraints. Another set of tools then translates models into executable specifications that capture the platform behavior, constraints, and interactions with the environment. These executable specifications in turn can be used to synthesize application software.

Earlier efforts at model-based development and code synthesis attempted by CASE tools generally failed to deliver on their potential for the following reasons [All02]:

- They attempted to generate entire applications, including the infrastructure and the application logic, which led to inefficient, bloated code that was hard to optimize, validate, evolve, or integrate with existing code.
- Due to the lack of sophisticated domain-specific languages and associated modeling tools, it was hard to achieve round-trip engineering, that is, moving back and forth seamlessly between model representations and the synthesized code.
- Since CASE tools and early modeling languages dealt primarily with a restricted set of platforms (such as mainframes) and legacy programming languages (such as COBOL), they did not adapt well to the distributed computing paradigm that arose

from advances in PC and Internet technology and newer object-oriented programming languages, such as Java, C++, and C#.

Many of the limitations with model-integrated computing outlined above can be overcome by integrating MIC tools and processes with object-oriented frameworks [GSNW02]. This integration helps to overcome problems with earlier-generation CASE tools since it does not require the modeling tools to generate all the code. Instead, large portions of applications can be *composed* from reusable, prevalidated framework classes. Likewise, integrating MIC with frameworks helps address environments where application requirements and functionality change at a rapid pace by synthesizing and assembling newer extended framework classes and automating the configuration of many QoS-critical aspects, such as concurrency, distribution, transactions, security, and dependability.

The combination of model-integrated computing with frameworks, components, and patterns is an area of active research [Bay02]. In the DOC group, for example, there are R&D efforts underway to develop a MIC tool suite called the *Component Synthesis with Model-Integrated Computing* (CoSMIC) [GSNW02]. CoSMIC extends the popular GME modeling and synthesis tools [LBM+01] and the ACE ORB (TAO) [SLM98] to support the development, assembly, and deployment of QoS-enabled networked applications. To ensure the QoS requirements can be realized in the middleware layer, CoSMIC's model-integrated computing tools can specify and analyze the QoS requirements of application components in their accompanying metadata.

1.3 Applying Frameworks to Network Programming

One reason why it's hard to write robust, extensible, and efficient networked applications is that developers must master many complex networking programming concepts and mechanisms, including

- Network addressing and service identification/discovery
- Presentation layer conversions, such as marshaling, demarshaling, and encryption, to handle heterogeneous hosts with alternative processor byte orderings
- Local and remote interprocess communication (IPC) mechanisms
- Event demultiplexing and event handler dispatching
- Process/thread lifetime management and synchronization

Application programming interfaces (APIs) and tools have evolved over the years to simplify the development of networked applications and middleware. Figure 1.6 illustrates the IPC APIs available on OS platforms ranging from UNIX to many real-time operating systems. This figure shows how applications can access networking APIs for local and remote IPC at several levels of abstraction. We briefly discuss each level of abstraction below, starting from the lower-level kernel APIs to the native OS user-level networking APIs and the host infrastructure middleware.

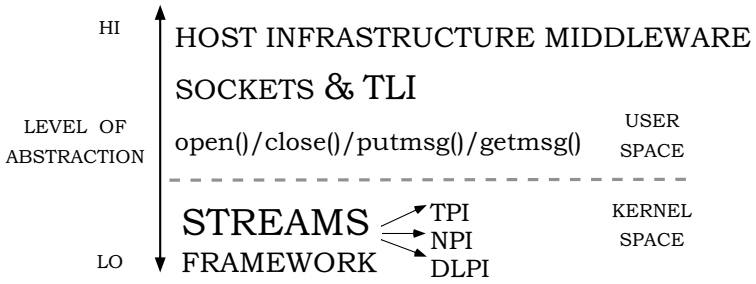

Figure 1.6: Levels of Abstraction for Network Programming

Kernel-level networking APIs. Lower-level networking APIs are available in an OS kernel's I/O subsystem. For example, the UNIX `putmsg()` and `getmsg()` system functions can be used to access the *Transport Provider Interface* (TPI) [OSI92b] and the *Data Link Provider Interface* (DLPI) [OSI92a] available in System V STREAMS [Rit84]. It's also possible to develop network services, such as routers [KMC+00], network file systems [WLS+85], or even Web servers [JKN+01], that reside entirely within an OS kernel.

Programming directly to kernel-level networking APIs is rarely portable between different OS platforms, however. It's often not even portable across different versions of the same OS! Since kernel-level programming isn't used in most networked applications, we don't cover it any further in this book. See [Rag93], [SW95, MBKQ96], and [SR00] for coverage of these topics in the context of System V UNIX, BSD UNIX, and Windows 2000, respectively.

User-level networking APIs. Networking protocol stacks in modern commercial operating systems reside within the protected address space of the OS kernel. Applications running in user space access protocol stacks in the OS kernel via IPC APIs, such as the Socket or TLI APIs. These APIs collaborate with an OS kernel to provide the capabilities shown in the following table:

Capability	Description
Local endpoint management	Create and destroy local communication endpoints, allowing access to available networking facilities.
Connection establishment and connection termination	Enable applications to establish connections actively or passively with remote peers and to shutdown all or part of the connections when transmissions are complete.
Options management	Negotiate and enable/disable protocol and endpoint options.
Data transfer mechanisms	Exchange data with peer applications.
Name/address translation	Convert human-readable names to low-level network addresses and vice versa.

These capabilities are covered in Chapter 2 of C++NPv1 in the context of the Socket API.

Many IPC APIs are modeled loosely on the UNIX file I/O API, which defines the `open()`, `read()`, `write()`, `close()`, `ioctl()`, `lseek()`, and `select()` functions [Rit84]. Due to syntactic and semantic differences between file I/O and network I/O, however, networking APIs provide additional functionality that's not supported directly by the standard UNIX file I/O APIs. For example, the pathnames used to identify files on a UNIX system aren't globally unique across hosts in a heterogeneous distributed environment. Different naming schemes, such as IP host addresses and TCP/UDP port numbers, have therefore been devised to uniquely identify communication endpoints used by networked applications.

Host infrastructure middleware frameworks. Many networked applications exchange messages using synchronous and/or asynchronous request/response protocols in conjunction with host infrastructure middleware frameworks. Host infrastructure middleware encapsulates OS concurrency and IPC mechanisms to automate many low-level aspects of networked application development, including

- Connection management and event handler initialization
- Event detection, demultiplexing, and event handler dispatching
- Message framing atop bytestream protocols, such as TCP
- Presentation conversion issues involving network byte ordering and parameter marshaling and demarshaling
- Concurrency models and synchronization of concurrent operations
- Networked application composition from dynamically configured services
- Hierarchical structuring of layered networked applications and services
- Management of quality of service (QoS) properties, such as scheduling access to processors, networks, and memory

The increasing availability and popularity of high-quality and affordable host infrastructure middleware is helping to raise the level of abstraction at which developers of networked applications can work effectively. For example, [C++NPv1, SS02] present an overview of higher-level distributed object computing middleware, such as CORBA [Obj02] and The ACE ORB (TAO) [SLM98], which is an implementation of CORBA built using the frameworks and classes in ACE. It's still useful, however, to understand how lower level IPC mechanisms work to fully comprehend the challenges that arise when designing, porting, and optimizing networked applications.

1.4 A Tour through the ACE Frameworks

1.4.1 An Overview of ACE

ACE is a highly portable, widely used, open-source host infrastructure middleware toolkit. The source code is freely available from `http://ace.ece.uci.edu/` or `http://`

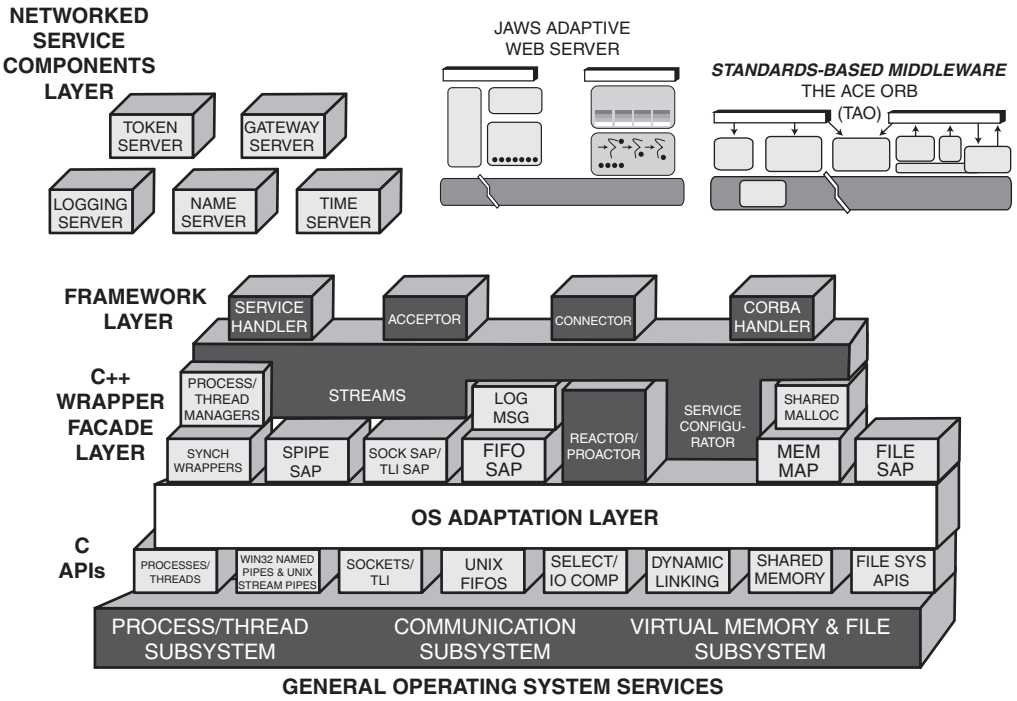

Figure 1.7: The Layered Architecture of ACE

`www.riverace.com/`. The core ACE library contains roughly a quarter million lines of C++ code that comprises approximately 500 classes. Many of these classes cooperate to form ACE's major frameworks. The ACE toolkit also includes higher-level components, as well as a large set of examples and an extensive automated regression test suite.

To separate concerns, reduce complexity, and permit functional subsetting, ACE is designed using a layered architecture [POSA1], shown in Figure 1.7. The capabilities provided by ACE span the session, presentation, and application layers in the OSI reference model [Bla91]. The foundation of the ACE toolkit is its combination of an OS adaptation layer and C++ wrapper facades, which together encapsulate core OS network programming mechanisms to run portably on all the OS platforms shown in Sidebar 2 (page 16). The higher layers of ACE build on this foundation to provide reusable frameworks, networked service components, and standards-based middleware.

1.4.2 A Synopsis of the ACE Frameworks

The ACE frameworks are an integrated set of classes that can be instantiated and customized to provide complete networked applications and service components. These frameworks help to transfer decades of accumulated knowledge directly from the ACE developers to

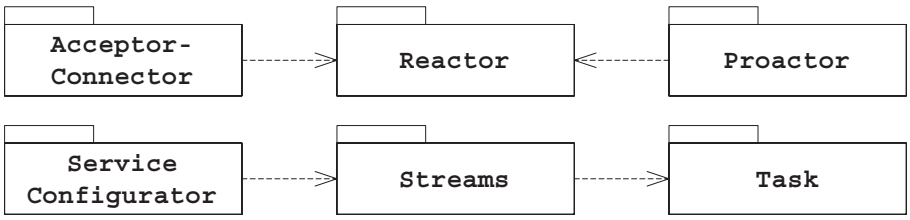

Figure 1.8: The Key Frameworks in ACE

Sidebar 2: OS Platforms Supported by ACE

ACE runs on a wide range of operating systems, including:

- PCs, for example, Windows (32- and 64-bit versions), WinCE, and Macintosh OS X
- Most versions of UNIX, for example, SunOS/Solaris, IRIX, HP-UX, Tru64 UNIX (Digital UNIX), AIX, DG/UX, Linux (Redhat, Debian, and SuSE), SCO OpenServer, UnixWare, NetBSD, and FreeBSD
- Real-time operating systems, for example, VxWorks, ChorusOS, LynxOS, Pharlap TNT, QNX Neutrino and RTP, RTEMS, and pSoS
- Large enterprise systems, for example, OpenVMS, MVS OpenEdition, Tandem NonStop-UX, and Cray UNICOS.

ACE can be used with all of the major C++ compilers on these platforms. The ACE Web site at `http://ace.ece.uci.edu` contains a complete, up-to-date list of platforms, along with instructions for downloading and building ACE.

ACE users in the form of expertise embodied in well-tested and reusable C++ software artifacts. The ACE frameworks implement a pattern language for programming concurrent object-oriented networked applications. Figure 1.8 illustrates the ACE frameworks. To illustrate how the ACE frameworks rely on and use each other, the lines between boxes represent a dependency in the direction of the arrow. Each framework is outlined below.

ACE Reactor and Proactor frameworks. These frameworks implement the Reactor and Proactor patterns [POSA2], respectively. Both are architectural patterns that allow applications to be driven by events that are delivered to the application from one or more event sources, the most important of which are I/O endpoints. The Reactor framework facilitates a *reactive I/O* model, with events signaling the ability to begin a synchronous I/O operation. The Proactor framework is designed for a *proactive I/O* model where one or more asynchronous I/O operations are initiated and the completion of each operation triggers an event. Proactive I/O models can achieve the performance benefits of concurrency without incurring many of its liabilities. The Reactor and Proactor frameworks automate the detection, demultiplexing, and dispatching of application-defined handlers in response to many

types of events. Chapters 3 and 4 describe the ACE Reactor framework and Chapter 8 describes the ACE Proactor framework.

ACE Service Configurator framework. This framework implements the Component Configurator pattern [POSA2], which is a design pattern that allows an application to link and unlink its component implementations without having to modify, recompile, or relink the application statically. The ACE Service Configurator framework supports the configuration of applications whose services can be assembled late in the design cycle, such as at installation time and/or run time. Applications with high availability requirements, such as mission-critical systems that perform online transaction processing or real-time industrial process automation, often require such flexible configuration capabilities. Chapter 2 describes the design dimensions associated with configuring networked services and Chapter 5 describes the ACE Service Configurator framework.

ACE Task framework. This framework implements various concurrency patterns, such as Active Object and Half-Sync/Half-Async [POSA2]. Active Object is a design pattern that decouples the thread that executes a method from the thread that invoked it. Its purpose is to enhance concurrency and simplify synchronized access to objects that reside in their own threads of control. Half-Sync/Half-Async is an architectural pattern that decouples asynchronous and synchronous processing in concurrent systems, to simplify programming without reducing performance unduly. This pattern incorporates two intercommunicating layers, one for asynchronous and one for synchronous service processing. A queueing layer mediates communication between services in the asynchronous and synchronous layers. Chapter 5 of C++NPv1 describes the design dimensions associated with concurrent networked applications and Chapter 6 of this book describes the ACE Task framework.

ACE Acceptor-Connector framework. This framework leverages the Reactor framework and reifies the Acceptor-Connector pattern [POSA2]. This design pattern decouples the connection and initialization of cooperating peer services in a networked system from the processing they perform once connected and initialized. The Acceptor-Connector framework decouples the active and passive initialization roles from application-defined service processing performed by communicating peer services after initialization is complete. Chapter 7 describes this framework.

ACE Streams framework. This framework implements the Pipes and Filters pattern, which is an architectural pattern that provides a structure for systems that process a stream of data [POSA1]. The ACE Streams framework simplifies the development and composition of hierarchically layered services, such as user-level protocol stacks and network management agents [SS94]. Chapter 9 describes this framework.

When used together, the ACE frameworks outlined above enable the development of networked applications that can be updated and extended without the need to modify, re-

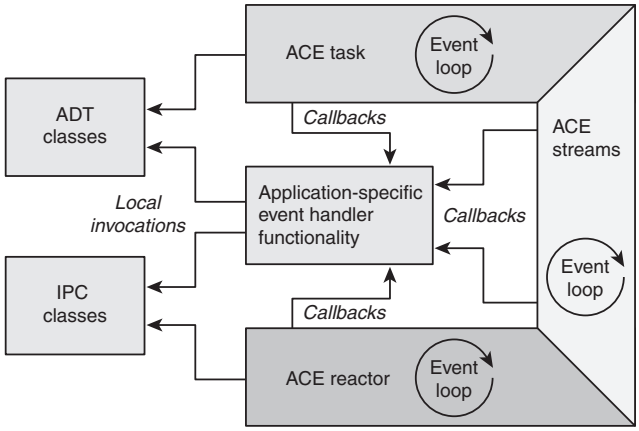

Figure 1.9: Applying Class Libraries to Develop and Use ACE Frameworks

compile, relink, or restart running applications. ACE achieves this unprecedented flexibility and extensibility by combining

- **OS mechanisms,** such as event demultiplexing, IPC, dynamic linking, multithreading, multiprocessing, and synchronization [Ste99]
- **C++ language features,** such as templates, inheritance, and dynamic binding [Bja00]
- **Patterns,** such as Component Configurator [POSA2], Strategy [GoF], and Handler/Callback [Ber95]

The ACE frameworks provide inversion of control via callbacks, as shown below:

ACE Framework	Inversion of Control
Reactor and Proactor	Calls back to application-supplied event handlers to perform processing when events occur synchronously and asynchronously.
Service Configurator	Calls back to application-supplied service objects to initialize, suspend, resume, and finalize them.
Task	Calls back to an application-supplied hook method to perform processing in one or more threads of control.
Acceptor-Connector	Calls back to service handlers to initialize them after they're connected.
Streams	Calls back to initialize and finalize tasks when they are pushed and popped from a stream.

The callback methods in ACE's framework classes are defined as C++ virtual methods. This use of dynamic binding allows networked applications to freely implement and extend interface methods without modifying or rebuilding existing framework classes. In contrast, the ACE wrapper facades rarely use callbacks or virtual methods, so they aren't as extensible as the ACE frameworks. The ACE wrapper facades do support a broad range of use

cases, however, and can be integrated together via generic programming [Ale01] techniques based on the C++ *traits* and *traits classes idioms* outlined in Sidebar 40 (page 165).

Figure 1.9 illustrates how the class libraries and frameworks in ACE are complementary technologies. The ACE toolkit simplifies the implementation of its frameworks via its class libraries of containers, which include lists, queues, hash tables, strings, and other reusable data structures. Likewise, application-defined code invoked by event handlers in the ACE Reactor framework can use the ACE wrapper facades and the C++ standard library classes [Jos99] to perform IPC, synchronization, file management, and string processing operations. Sidebar 3 describes how to build the ACE library so that you can experiment with the examples we present in this book.

Sidebar 3: Building ACE and Programs that Use ACE

ACE is open-source software that you can download from `http://ace.ece.uci.edu` or `http://www.riverace.com` and build yourself. These sites contain a wealth of other material on ACE, such as tutorials, technical papers, and an overview of other ACE wrapper facades and frameworks that aren't covered in this book. You can also purchase a prebuilt version of ACE from Riverace at a nominal cost. See `http://www.riverace.com` for a list of the prebuilt compiler and OS platforms supported by Riverace.

If you want to build ACE yourself, you should download and unpack the ACE distribution into an empty directory. The top-level directory in the distribution is named `ACE_wrappers`. We refer to this top-level directory as "`ACE_ROOT`." You should create an environment variable by that name containing the full path to the top-level ACE directory. The ACE source and header files reside in `$ACE_ROOT/ace`.

The `$ACE_ROOT/ACE-INSTALL.html` file has complete instructions for building ACE, including how to configure it for your OS and compiler. This book's networked logging service example source and header files reside in `$ACE_ROOT/examples/C++NPv2` and are ready to build on all platforms that ACE supports. To build your own programs, the `$ACE_ROOT` directory must be added to your compiler's file include path. For command-line compilers, this can be done with the `-I` or `/I` compiler option. Graphical IDEs provide similar options, such as MSVC++'s "Preprocessor, Additional include directories" section of the C/C++ tab on the Project Settings dialog box.

1.5 Example: A Networked Logging Service

It's been our experience that the principles, methods, and skills required to develop and use reusable networked application software cannot be learned solely by generalities or toy examples. Instead, programmers must learn concrete technical skills and gain hands-on experience by developing and using real frameworks and applications. We therefore

illustrate key points and ACE capabilities throughout this book by extending and enhancing the networked logging service example introduced in C++NPv1, which collects and records diagnostic information sent from one or more client applications.

The logging service in C++NPv1 used many of ACE's wrapper facades in a two-tier client/server architecture. This book's logging service examples use a more powerful architecture that illustrates a broader complement of capabilities and patterns, and demonstrates how ACE's frameworks can help achieve efficient, predictable, and scalable networked applications. This service also helps to demonstrate key design and implementation considerations and solutions that will arise when you develop your own concurrent object-oriented networked applications.

Figure 1.10 illustrates the application processes and daemons in our networked logging service, which we outline below.

Client application processes (such as P_1, P_2, and P_3) run on client hosts and generate log records ranging from debugging messages to critical error messages. The logging information sent by a client application contains the time the log record was created, the process identifier of the application, the priority level of the log record, and a variable-sized string containing the log record text message. Client applications send these log records to a *client logging daemon* running on their local host.

Client logging daemons run on every host machine participating in the networked logging service. Each client logging daemon receives log records from that host's client applications via some form of local IPC mechanism, such as shared memory, pipes, or sockets. The client logging daemon uses a remote IPC mechanism, such as TCP/IP, to forward log records to a *server logging daemon* running on a designated host.

Server logging daemons collect and output the incoming log records they receive from client applications via client logging daemons. A server logging daemon[2] can determine which client host sent each message by using addressing information it obtains from the underlying Socket API. There's generally one server logging daemon per system configuration, though they could be replicated to avoid a single point of failure.

Figure 1.11 (page 22) shows the progression of networked application servers that we'll develop and use in this book. These client and server logging daemons will illustrate how to use the ACE frameworks and wrapper facades with the following concurrency models.

Concurrency Model	Section
Reactive	3.5, 4.2, 5.4
Thread pool	4.3, 4.4, 6.3
Thread-per-connection	7.2, 7.3
Producer/consumer	6.2, 7.4, 9.2
Proactive	8.2 – 8.5

[2] We use the terms *server logging daemon* and *logging server* interchangeably throughout this book.

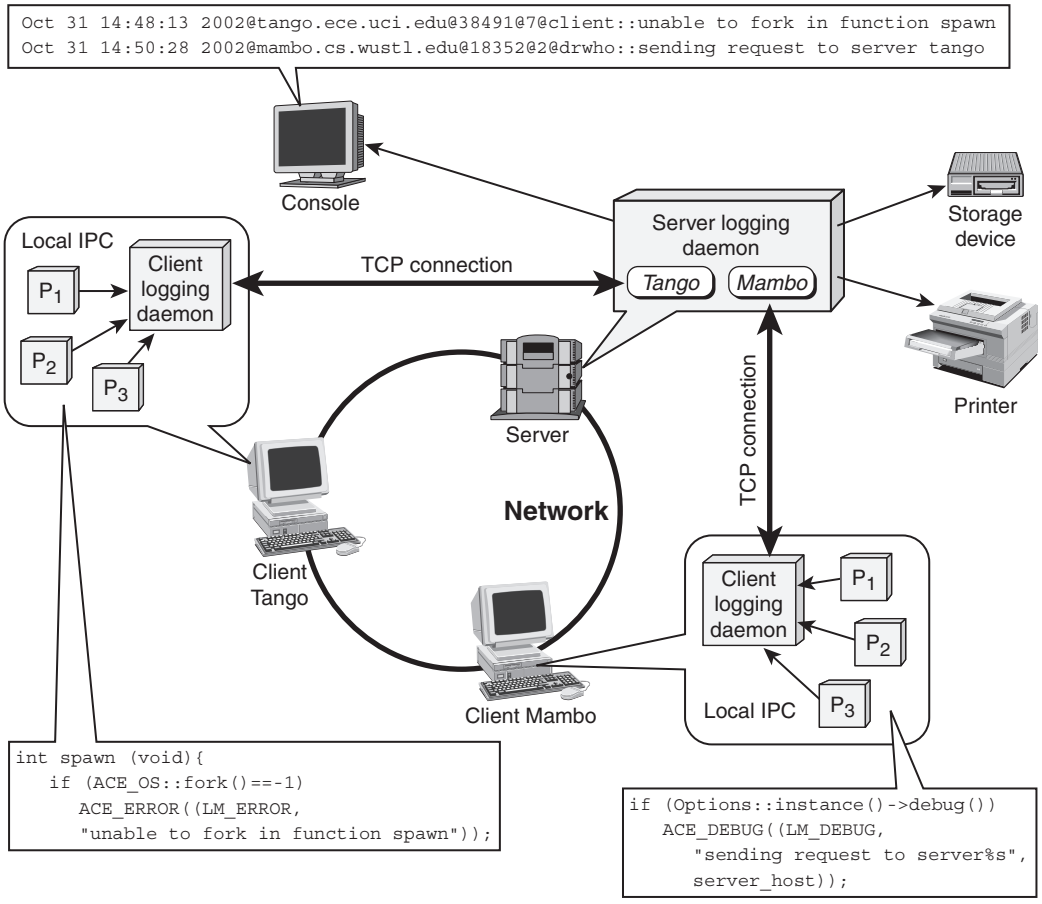

Figure 1.10: Processes and Daemons in the Networked Logging Service

1.6 Summary

Networked application software has been developed manually from scratch for decades. The continual rediscovery and reinvention of core concepts and capabilities associated with this process has kept the costs of engineering and evolving networked applications too high for too long. Improving the quality and quantity of systematic software reuse is essential to resolve this problem.

Middleware is a class of software that's particularly effective at providing systematically reusable artifacts for networked applications. Developing and using middleware is therefore an important way to increase reuse. There are many technical and nontechnical challenges that make middleware development and reuse hard, however. This chapter described how object-oriented frameworks can be applied to overcome many of these chal-

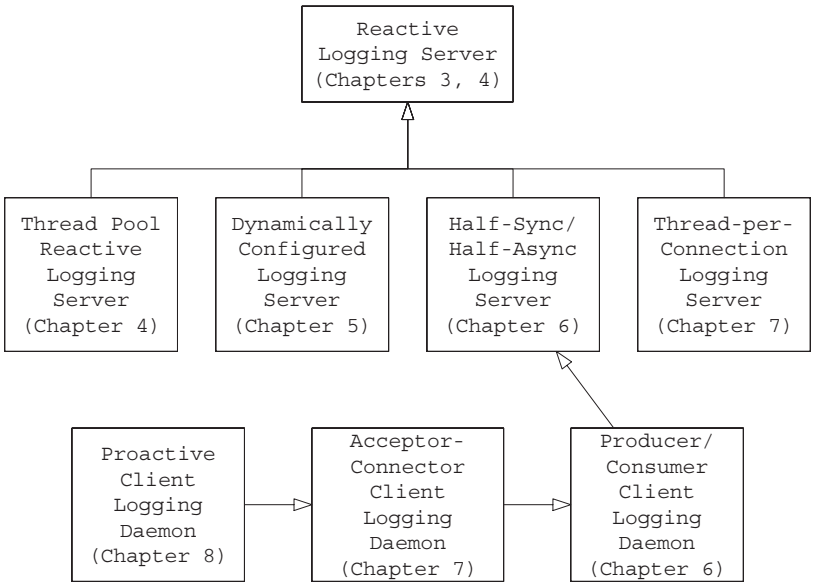

Figure 1.11: Logging Server Examples

lenges. To make the most appropriate choice of software development technologies, we also described the differences between frameworks and class libraries, components, patterns, and model-integrated computing. Each technology plays a part in reducing software development costs and life cycles and increasing software quality, functionality, and performance.

The result of applying framework development principles and patterns to the domain of networked applications has yielded the ACE frameworks. These frameworks handle common network programming tasks and can be customized via C++ language features to produce complete networked applications. When used together, the ACE frameworks simplify the creation, composition, configuration, and porting of networked applications without incurring significant performance overhead. The rest of this book explains how and why the ACE frameworks were developed and shows many examples of how ACE uses C++ features to achieve its goals.

An intangible, but valuable, benefit of ACE is its transfer of decades of accumulated knowledge from ACE framework developers to ACE framework users in the form of expertise embodied in well-tested C++ classes that implement time-proven networked application software development strategies. These frameworks took scores of person-years to develop, optimize, and mature. Fortunately, you can take advantage of the expertise embodied in these frameworks without having to independently rediscover or reinvent the patterns and classes that underlie them.