# Why Standards Alone Won't Get You Portable Software
# And How to Make Open Source Development Work for You

**Douglas C. Schmidt and Stephen D. Huston**

## 1. Motivation

The need to write portable software that runs on a variety of computing platforms becomes more obvious every day. Leading mainstream computer vendors, such as IBM, HP, Compaq, and Dell, offer a mix of Windows, Linux, and UNIX operating systems across their hardware platforms. Likewise, as people become ever more connected and mobile, many computer vendors are also supporting embedded and handheld systems.

As a software professional, it's your job to develop software that enables your company to gain competitive advantage. The key to that advantage often lies in creating portable software that runs on multiple platforms, and versions of platforms. If you believe the talk in some software development circles, you might think that *de facto* standards, such as Windows, or *de jure* standards, such as POSIX and UNIX98, are all you need to make your applications portable across the growing variety of computing platforms outlined above. Unfortunately, the old adage that "the nice thing about standards is that there are so many to choose from" is even more applicable today than it was a decade ago. There are now dozens of different operating system (OS) platforms used in commercial, academic, and government projects including real-time, embedded, and handheld systems; personal and laptop computers; an assortment of various-sized UNIX or Linux systems; and "big iron" mainframes and even supercomputers. Moreover, the number of OS permutations grows with each new version and variant.

In theory, the idea behind standards is sound: if a standard is implemented by many vendors (or one uber vendor), code that adheres to the standard will work on all platforms that implement the standard. In practice, however, standards evolve and change, just like software. Moreover, vendors often choose to implement different standards at different times. It's therefore likely that you'll work on multiple computing platforms that implement different standards in different ways at different times.

Since your customers pay you to solve their business needs—not to wrestle continuously with portability details—it's worthwhile to consider how to ensure that standards work for you, rather than against you. To assist you in this quest, this article describes some difficulties you're likely to encounter when relying on standards—OS standards in particular—for portability. It then describes some of the ways that middleware and open-source software models can help you develop portable networked applications more quickly and easily.

## 2. Problem: Programming Yourself into a Corner with OS APIs

An OS can be viewed as a "silicon abstraction layer," which shields application software from the details of the underlying hardware. If just one instance of one version of an OS application programming interface (API) were adopted universally, our programming tasks would clearly be simplified. As noted above, however, that's not the case today, nor will it ever be due to the need to support legacy applications and to advance technology. As a result, the common practice of programming applications directly to OS APIs yields the following problems:

- **It's not portable.** Some standards are implemented on only a subset of your target platforms. For example, three popular threading APIs are Windows, UNIX International (UI) threads, and POSIX threads (Pthreads). They all have different features and semantics, and of course, different APIs. If your code needs to run on Windows and anything else that isn't Windows, therefore, you'll need to deal with at least two of these three APIs. To make matters worse, these APIs have evolved over time, so code written to an earlier version of the API may not compile with later versions, or worse it may compile but behave differently! If the source code for OS APIs isn't available, you can't adjust it to achieve backwards compatibility. Likewise, trying to integrate your own versions of vendor-supplied libraries is problematic, even if you could find a way to obtain the source code.
- **The differences are tedious to find and work around.** For instance, the enormously popular BSD Socket API is used for TCP/IP network programming. It's widely implemented and you can usually count on it being available on any platform that supports the TCP/IP networking protocols. However, the integration of the Socket API into the operating system's runtime libraries can yield functionality that's not portable. For example, although the

Socket API defines the `send()` and `recv()` functions to send and receive socket data, the `read()` and `write()` system calls on many UNIX systems can also be used for the same purpose, which allows simple substitution of another IPC mechanism (for example, pipes) for sockets. However, code written to take advantage of this feature won't work on systems where the Socket API is not closely united with other I/O subsystems, as you've no doubt noticed if you've tried to port UNIX-based Socket applications to other operating systems, such as Windows or some real-time systems.

- **It's error prone** since native OS APIs written in C often lack type safe, reentrant, and extensible system function interfaces and function libraries. For example, endpoints of communication in the Socket API are identified via weakly typed integer or pointer I/O handles. Weakly typed handles increase the likelihood of subtle programming errors that don't surface until run time, which can cause serious problems for your customers.
- **It encourages inadequate design techniques** since many applications written using OS APIs are based on algorithmic design, rather than object-oriented design. Algorithmic design decomposes the structure of an application according to specific functional requirements, which are volatile and likely to evolve over time. This design paradigm therefore yields nonextensible software architectures that can't be customized rapidly to meet changing application requirements.

The bottom line is that it's easy to program yourself into a corner by developing applications entirely from scratch using nonportable native OS APIs and algorithmic design techniques. In this age of economic upheaval, deregulation, and stiff global competition, it's prohibitively expensive and time consuming for companies to make these mistakes, particularly when there's a better way!

### 3. An Appealing Solution: Host Infrastructure Middleware

An increasingly popular solution to the problems described above is to interpose host infrastructure middleware between OS APIs and application software. Host infrastructure middleware provides an “OS abstraction layer” that shields application software from the details of the underlying OS. Widely used examples of host infrastructure middleware include the following.

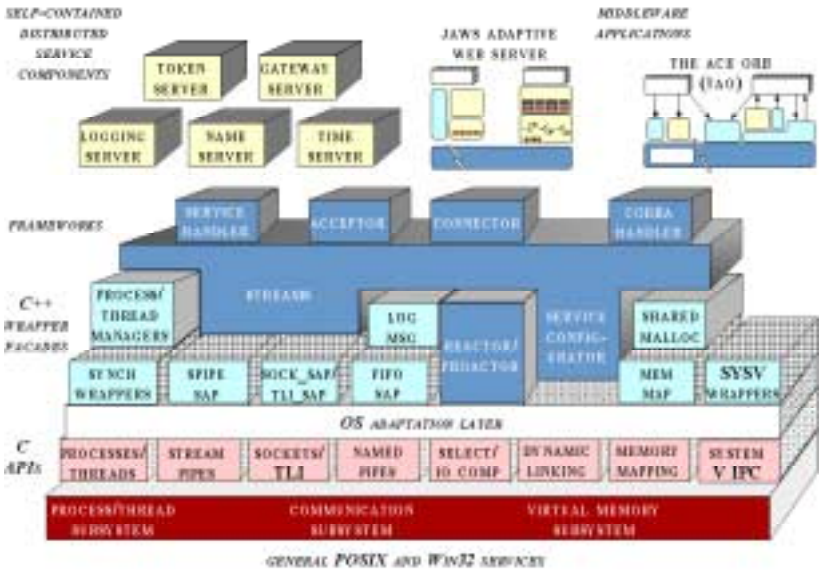
- The Sun Java Virtual Machine (JVM), which provides a platform-independent way of executing code by abstracting the differences between operating systems and CPU architectures. A JVM is responsible for interpreting Java bytecode and for translating the bytecode into an action or operating system call. It's the JVM's responsibility to encapsulate platform details within the portable bytecode interface, so that applications are shielded from disparate operating systems and CPU architectures on which Java software runs.
- The Microsoft Common Language Runtime (CLR), which is the host infrastructure middleware foundation upon which Microsoft's .NET web services are built. The Microsoft CLR is similar to Sun's JVM. For example, it provides an execution environment that manages running code and simplifies software development via automatic memory management mechanisms, cross-language integration, interoperability with existing code and systems, simplified deployment, and a security system.
- The ADAPTIVE Communication Environment (ACE), which is a freely available, open source, highly portable toolkit written in C++ that shields applications from differences between native OS programming capabilities, such as file handling, connection establishment, event demultiplexing, interprocess communication, (de)marshaling, concurrency, and synchronization. At the core of ACE is its OS adaptation layer and C++ wrapper facades that encapsulate OS file system, concurrency, and network programming mechanisms. The higher layers of ACE build upon this foundation to provide reusable frameworks that handle network programming tasks, such as synchronous and asynchronous event handling, service configuration and initialization, concurrency control, connection management, and hierarchical service integration.

  The primary differences between ACE, JVMs, and the .NET CLR are that (1) ACE is always a compiled C++ interface, rather than an interpreted bytecode interface, which removes a level of indirection and helps to optimize runtime performance, (2) ACE is open source, so it's possible to subset it or modify it to meet a wide variety of needs, and (3) ACE runs on more OS and hardware platforms than JVMs and CLR, including
  - **PCs**, for example, Windows (all 32/64-bit versions), WinCE; Redhat, Debian, and SuSE Linux; and Macintosh OS X.

- **Most versions of UNIX**, for example, SunOS 4.x and Solaris, SGI IRIX, HP-UX, Digital UNIX (Compaq Tru64), AIX, DG/UX, SCO OpenServer, UnixWare, NetBSD, and FreeBSD.
- **Real time operating systems**, for example, VxWorks, OS/9, Chorus, Integrity, LynxOS, Pharlap TNT, QNX Neutrino and RTP, RTEMS, and pSoS.
- **Large enterprise systems**, for example, OpenVMS, MVS OpenEdition, Tandem NonStop-UX, and Cray UNICOS.

### 4. Improving Application Portability with ACE

ACE contains ~250,000 lines of C++ code, ~500 classes, and ~10 frameworks **[ACE]**. To provide its powerful capabilities across a diverse range of platforms, ACE is designed using a layered architecture shown in the following figure.




This design allows networked application developers a wide variety of usage options to match their needs. It also makes reuse much simpler within ACE itself. The layers in ACE that enable these capabilities are described below.

- **ACE's OS adaptation layer.** This layer does most of the work to unify the diverse set of supported platforms and standards under a common API. In most cases, the OS adaptation layer presents a POSIX-like interface using efficient C++ inlined static methods. It's possible to write many networked applications portably using only ACE's OS adaptation layer. Other examples of OS adaptation layers include the Apache Portable Runtime and the Netscape Portable Runtime, which are open-source C libraries that form a system portability layer to many operating systems.

  Since an OS adaptation layer presents a flat C-like function API, however, programming to it directly incurs the drawbacks of algorithmic design. Moreover, this layer's purpose is to unify the means for implementing well-defined behavior, such as opening a file, writing data to a socket, or spawning a thread. Behaviors that aren't common across platforms, such as forking a process (not available on Windows, for example) are not implemented at this layer. For these reasons, many networked applications use ACE's wrapper facade layer, which is described next.

- **ACE's wrapper facade layer.** This layer provides an object-oriented form of systems programming for networked applications. Its classes reify the Wrapper Facade pattern **[POSA2]**, where one or more classes enforce proper usage by encapsulating functions and data within a type-safe OO interface, rather than a flat C function API. For example, there are separate wrapper facade classes that passively listen for TCP connections, actively connect a TCP connection, and send/receive data over the resulting active TCP streams. This layer also contains portable capabilities that don't share portable implementation details, such as process management.

  The ACE wrapper facade layer resolves many portability issues described in Section 2. The vast majority of portable functionality is available with the same classes across all supported platforms. Since ACE is responsible for resolving OS differences, your application code remains clean and neat on all platforms. Since ACE offers its functionality via

intelligently and efficiently designed class interfaces, your code won't suffer from either hard-to-find semantic differences across platforms or type-safety problems that can plague you at runtime. The result is that your code is both developed quicker, and the porting effort is much shorter than it would otherwise be.

As we've seen, ACE's OS adaptation and wrapper facade layers offer enormous benefit over trying to code directly to native OS APIs. However, ACE offers many more benefits than just object-oriented portability. These benefits are derived from the ACE framework layer described next.

- **ACE framework layer.** A framework is a set of classes that collaborates to provide a reusable architecture for a family of related applications **[Frameworks]**. The framework layer is the highest level of abstraction in the ACE toolkit. This layer codifies the interaction of the ACE wrapper facades to offer capabilities such as event demultiplexing and dispatching, asynchronous I/O handling, and event logging. Although implementing the ACE frameworks efficiently sometimes necessitates the use of platform-specific details, the ACE framework interfaces expose their capabilities as portably as possible via their systematic use of patterns **[POSA2]**.

  ACE's frameworks also offer a set of "semi-complete" applications, such as the Acceptor-Connector framework, which simplifies the active and passive establishment of network service sessions, and the Reactor framework, which detects and demultiplexes events from multiple sources and dispatches the events to application-defined event handlers that process the events. Developing complete applications via these frameworks just requires adding business-specific logic in well-defined locations, known as "hook methods", rather than recreating core areas of needed functionality. Framework-based development thus makes it easy to prototype and enhance designs rapidly because the architecture of many applications is embodied in ACE's frameworks.

  In addition to developing networked applications, ACE's frameworks have been used to develop even higher levels of standards-based middleware, such as the JAWS adaptive web server **[POSA2]** and The ACE ORB (TAO) **[TAO]**.

Together, the ACE middleware layers described above simplify the creation, composition, configuration, and porting of networked applications without incurring significant performance overhead. ACE's layered architecture also allows code to be migrated from simpler designs and prototypes to more complex and complete ones. This benefit is particularly important in large projects where requirements evolve and changes occur in a product, and where developers gain more experience with the ACE toolkit.

## 5. The Importance of Open Source

Open-source development processes have emerged as an effective approach **[OpenSource]** to reduce cycle time and decrease design, implementation, and quality assurance costs for certain types of software, particularly systems infrastructure software, such as operating systems, compilers and language processing tools, editors, and middleware. This section describes the reasons why successful open-source development projects work, from both an end user and software process perspective. We base this discussion on our many years of experience researching, devising, and employing open-source development processes and middleware toolkits, such as ACE, JAWS, and TAO.

From an end user perspective, successful open-source projects work for the following reasons:

- **Reduced software acquisition costs.** Open-source software is often distributed without development or run time license fees, though many open-source companies do charge for technical support. This pricing model is particularly attractive to application developers working in highly commoditized markets where profits are driven to marginal cost. Moreover, open-source projects typically use low-cost, widely accessible distribution channels, such as the Internet, so that users can access source code, examples, regression tests, and development information cheaply and rapidly.

- **Enhanced diversity and scale.** Well-written and well-configured open-source software can be ported easily to a variety of heterogeneous operating system and compiler platforms. In addition, since the source is available, end users have the freedom to modify and adapt their source base readily to fix bugs quickly or to respond to new market opportunities with greater agility. Indeed, many of the ACE ports originated in the ACE user community rather than its core development group. Due to ACE's open-source model, therefore, its range of platforms expanded rapidly.

- **Simplified collaboration.** Open-source promotes the sharing of programs and ideas among members of technology communities that have similar needs, but who also may have diverse technology acquisition and

funding strategies. This cross-fertilization can lead to new insights and breakthroughs that would not have occurred as readily without these collaborations. For example, due to input and code contributions from the ACE community, ACE's logging service component has evolved from a self-contained client/server arrangement into one that can take advantage of both UNIX syslog and the Windows Event Log for better enterprise integration.

From a software process perspective, successful open-source projects work for the following reasons:

- **Scalable division of labor.** Open-source projects work by exploiting a loophole in "Brooks Law" that states "adding developers to a late project makes it later." The logic underlying this law is that software development productivity generally doesn't scale up as the number of developers increases. The culprit is the rapid increase in human communication and coordination costs as project size grows. Thus, a team of 10 good developers can often produce much higher quality software with less effort and expense than a team of 1,000 developers.

  In contrast, software debugging and QA productivity can scale up as the number of developers helping to debug the software increases. This is particularly true in open-source development projects where the user community engages in very large-scale, distributed testing. For example, ACE's users are fairly autonomous and therefore don't need the level of coordination and communication that a conventional development team requires. ACE has an extensive set of regression tests, but its large user community brings many privately developed applications to bear on any new release, effectively multiplying the regression test suite many times over. Thus, in the testing/debugging arena, a team of 1,000 testers will usually find many more bugs than will a team of 10 testers. Due to this multiplied testing effect, the error-legs in open-source software are often identified quickly. The errors are also addressed very quickly because of a key advantage of open-source development discussed in the following bullet.

- **Short feedback loops.** One reason for the success of well-organized open-source development efforts is the short feedback loops between the core developers and the users. In ACE, for instance, it's often only a matter of minutes or hours from the point at which a bug is reported from a user to the point at which an official patch is supplied to fix it. Since many ACE users are also highly talented developers, users who encounter bugs often fix them directly and contribute the fixed source code back. Even if they can't provide immediate fixes, they can often provide concise test cases that allow the core ACE developers to isolate and resolve problems quickly.

  Short feedback loops in the ACE development process are also facilitated by its use of powerful Internet-enabled software configuration management tools, such as the GNU Concurrent Versioning System (CVS), the CVSup and viewCVS web interfaces to CVS, and the Bugzilla issue tracking database. These tools allow ACE users to synchronize in real time with updates supplied to and from the core ACE developers. User efforts greatly magnify the debugging and computing resources available to the ACE project, which in turn helps improves software quality.

- **Effective leverage of user community expertise and computing resources.** In today's time-to-market-driven economy, fewer software providers can afford long QA cycles. As a result, nearly everyone who uses a computer—particularly software application developers—is a beta tester of software that was shipped before all its defects were removed. In traditional closed-source/binary-only software deployment models, premature release cycles yield frustrated users with little recourse when problems arise with software they purchased from vendors and thus little incentive to help improve closed-source products. In contrast, open-source development processes help to leverage expertise and resources in their communities, thereby allowing core developers and users to collaborate to improve software quality. Due to the short feedback loops described above, open source users are rewarded by rapid fixes after bugs are identified.

  Particularly important to multiplatform software like ACE is the diversity of computing platforms that are tested simultaneously by the user community. In addition to the multiplication of test cases and scenarios mentioned above, the open-source nature of ACE leverages an enormous combination of hardware platforms, operating system versions, and compiler releases. The combination of expertise and computing resources has helped to make ACE successful across a wide range of platforms.

- **Inverted stratification.** In many organizations, testers are perceived to have less status than software developers. The open-source development model inverts this stratification in many cases so that "testers" among the user community are often excellent software application developers. These tester/developers can use their considerable debugging skills when they encounter occasional problems with the open-source software base. The open-source model makes it possible to employ the talents of these gifted developers, who would rarely be satisfied with

playing the role of a tester in traditional software organizations. In many open-source projects, such as ACE, the names of users who contribute fixes, ideas, and features are recorded in ChangeLog and THANKS files. Prolific contributors therefore achieve a level of international recognition in the community that can be highly rewarding.

In general, traditional closed-source software development and QA processes rarely achieve the benefits outlined above as rapidly or as cost effectively as open-source processes.

**6. Concluding Remarks**

Contemporary business climates require software developers to move applications nimbly among a wide variety of hardware and software platforms. The standards developed by vendors and industry groups to ease the burden of porting software often present a myriad of conflicting and divergent facilities and APIs, making it unnecessarily hard to develop portable software. Developers deserve better.

Host infrastructure middleware is an emerging set of technologies that helps alleviate many of the portability challenges with OS APIs in order to develop efficient, yet reusable and retargetable, networked applications quicker and easier. The ACE toolkit is host infrastructure middleware provides a layered and efficient set of classes and frameworks based on patterns to alleviate the problems of developing portable software across a wide range of platforms. ACE's open source roots have catalyzed its growth since its inception over a decade ago. After Doug Schmidt began developing ACE in 1991, over 1,400 contributors from more than 50 countries have contributed to ACE. This wide range of shared expertise and resources enables ACE's portability to extend across the world's most popular general-purpose and real-time operating systems. Today, ACE is being used by thousands of development teams, ranging from large Fortune 500 companies to small startups.

ACE has changed the way complex networked applications and middleware are being designed and implemented. Its open-source development model and self-supporting user community culture is similar in spirit and enthusiasm to that driving Linus Torvalds's popular Linux OS. The ACE developer and user community have been instrumental in transitioning ACE from the personal hobby of a single researcher into one of the world's most widely used C++ frameworks for concurrent object-oriented network programming across a wide range of hardware and software platforms.

**References**

**[ACE]** D. Schmidt and S. Huston, C++ Network Programming, Volume 1: Mastering Complexity with ACE and Patterns, Addison-Wesley, 2002.

**[Frameworks]** Building Application Frameworks: Object-Oriented Foundations of Framework Design, Wiley and Sons, 1999 (Mohamed Fayad and Ralph Johnson and Douglas C. Schmidt eds.).

**[OpenSource]** E. Raymond, The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary, O'Reilly, 2001.

**[POSA2]** D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Wiley and Sons, 2000.

**[TAO]** D. Schmidt, D. Levine, and S. Mungee, "The Design and Performance of the TAO Real-Time Object Request Broker", Computer Communications Special Issue on Building Quality of Service into Distributed Systems, 21(4), 1998.