



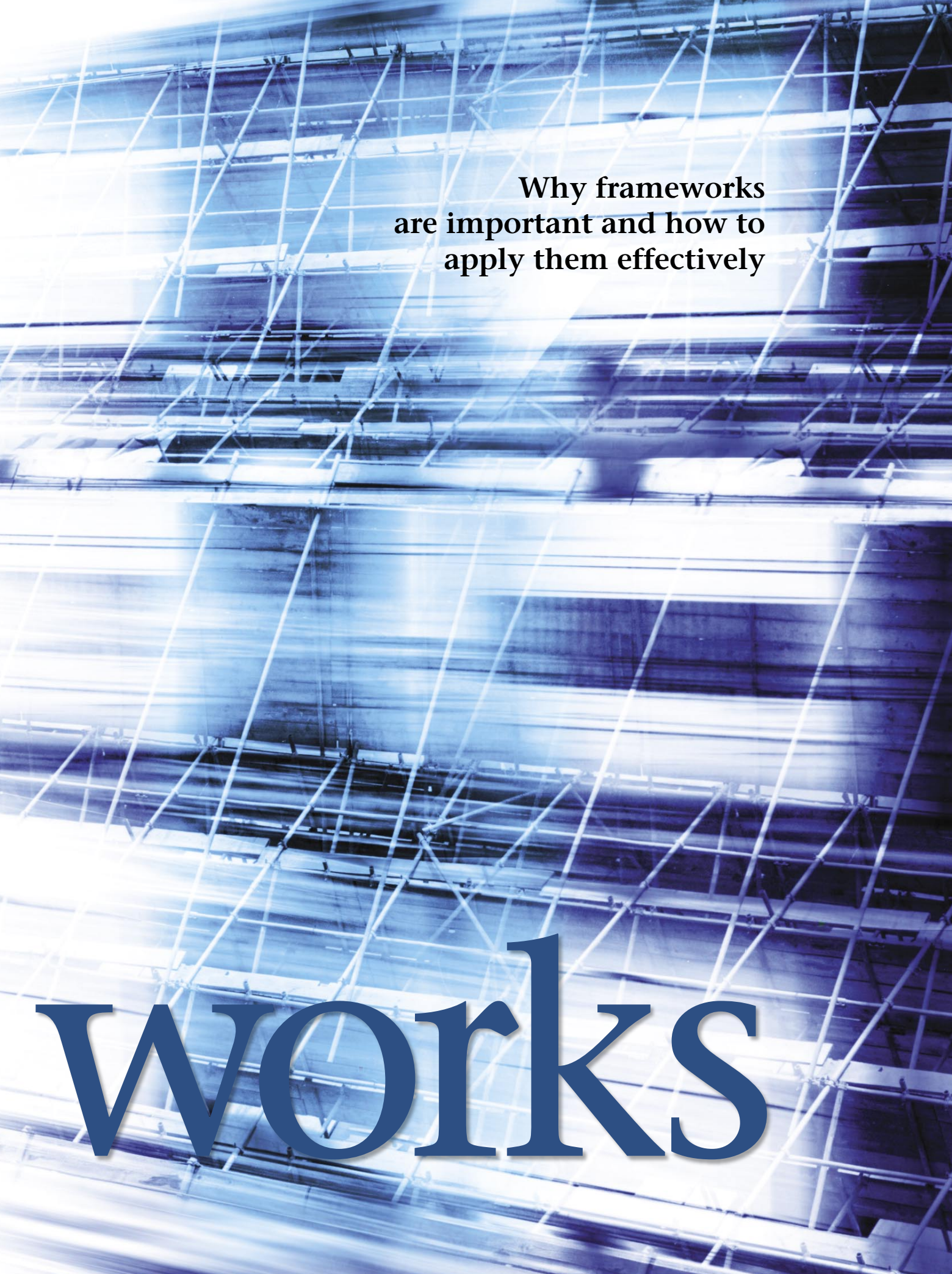
DOUGLAS C. SCHMIDT, ANIRUDDHA GOKHALE,
AND BALACHANDRAN NATARAJAN,
VANDERBILT UNIVERSITY

In today's competitive, fast-paced computing industry, successful software must increasingly be: (1) *extensible* to support successions of quick updates and additions to address new requirements and take advantage of emerging markets; (2) *flexible* to support a growing range of multimedia data types, traffic flows, and end-to-end QoS (quality of service) requirements; (3) *portable* to reduce the effort required to support applications on heterogeneous operating-system platforms and compilers; (4) *reliable* to ensure that applications are robust and tolerant to faults; (5) *scalable* to enable applications to handle larger numbers of clients simultaneously; and (6) *affordable* to ensure that the total ownership costs of software acquisition and evolution are not prohibitively high.

Achieving these qualities is difficult, however, when:

1. Core concepts and software artifacts are continually rediscovered and reinvented—that is, when the same functionality is rewritten and revalidated. Application software has historically been developed largely from scratch. This development process has been applied many times in many companies, by many projects and programmers in parallel. Even worse, it has been applied by the same teams in a series of projects. Regrettably, this continuous rediscovery and reinvention of core concepts and code has kept costs high and quality low throughout the software development life cycle. These

Leveraging Application Frame



Why frameworks
are important and how to
apply them effectively

works

Leveraging Application Frameworks

problems only get worse as hardware, networks, operating systems, middleware, and compilers continue to evolve. This “infrastructure churn” keeps shifting the foundations of application software development, resulting in a major source of *accidental complexity*, which arises from limitations with tools and techniques used to develop software.¹

2. Software is developed monolithically—as tightly coupled clumps of functionality that are not organized modularly. The functions in monolithic software are often tightly coupled via shared, global variables, and diagrams of their control flow often look like spaghetti. Monolithic software is therefore unnecessarily hard to understand, maintain, and extend.² Although monolithic software may sometimes be appropriate in short-lived, throw-away prototypes³ written by a single programmer, it is poorly suited for applications that must be maintained and enhanced by multiple developers over longer amounts of time.

To avoid the traps and pitfalls of writing and maintaining monolithic software, a more effective way to achieve quality software is to use *frameworks*.^{4,5} A framework is an integrated set of software artifacts (such as classes, objects, and components) that collaborate to provide a reusable architecture for a family of related applications.⁶ In particular, frameworks decouple the application-dependent portions of software from the application- and platform-independent portions, thereby enhancing software extensibility, flexibility, and portability in the following ways:

Design reuse—for example, by guiding application developers through the steps necessary to ensure successful creation and deployment of complex software.

Implementation reuse—for example, by amortizing software life-cycle costs and leveraging previous development and optimization effort.

Validation reuse—for example, by amortizing the effort of validating the application- and platform-independent portions of software, thereby enhancing software reliability and scalability.

Likewise, as frameworks mature and become commoditized in the form of COTS (commercial off-the-shelf)

products, they often become more affordable.

Although frameworks can be a powerful means to reduce software cost and improve its quality, they can be hard to understand, select, learn, use, debug, and optimize. To help make it easier to apply frameworks in practice, this article examines key characteristics that underlie various types of frameworks and explores key challenges that arise when developing and reusing frameworks. It then describes specific steps to address these challenges.

KEY CHARACTERISTICS OF FRAMEWORKS

Although frameworks are used in a wide range of different domains—such as telecommunications, avionics, manufacturing, and financial services—they share certain defining characteristics.⁶ Figure 1 illustrates three of the most important characteristics of frameworks that help them achieve the qualities outlined at the beginning of this article. These three characteristics are as follows:

A framework exhibits “inversion of control” at runtime via callbacks. These callbacks invoke the *hook methods* of application-defined components after the occurrence of

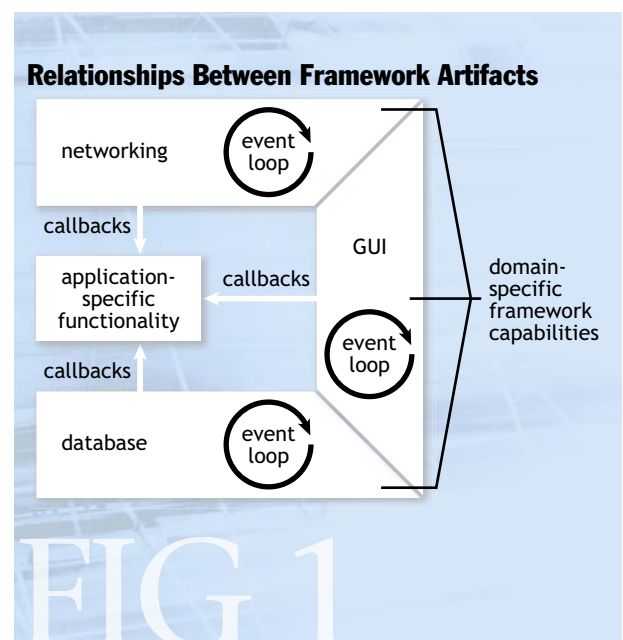


FIG 1

an event, such as a mouse click or data arriving on a network connection. When an event occurs, the framework calls back to a virtual hook method in a preregistered application component, which then performs application-defined processing in response to the event. The hook methods in the components decouple the application software from the reusable framework software, which allows each to change independently as long as the interface signature and interaction protocols are not modified. 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 the events.

A framework provides an integrated set of domain-specific structures and functionality based on patterns. Patterns codify reusable design expertise that provides time-proven solutions to commonly occurring software problems that arise in particular contexts and domains.⁷ Frameworks can be thought of as concrete realizations of groups of related patterns (known as *pattern languages*) that enable reuse of code by capturing the common abstractions of an application domain—both their structures and behaviors—while yielding control of application-specific structure and behavior to application developers. Frameworks reify the key roles, relationships, and patterns of interactions among software components in application domains as reusable code. They therefore can increase the amount of software reused, which in turn helps to reduce dramatically the amount of new software that is (re)written, debugged, and maintained.

A framework is a semi-complete application. Developers form complete applications by extending and customizing reusable components in the framework. In particular, frameworks help abstract common flows of control within applications in a domain into product-line architectures and families of related components. At runtime these components can collaborate to integrate customizable application-independent reusable code with customized application-defined code. Since a framework is a semi-complete application, it enables larger-scale reuse of software than can be achieved by reusing individual components or stand-alone functions.

Developers in certain domains have applied frameworks successfully for several decades. For example, early frameworks, such as MacApp, X Windows, and Interviews, originated in the domain of GUIs (graphical user interfaces). JFCs (Java Foundation Classes), MFCs (Microsoft Foundation Classes), and Qt are contemporary

GUI frameworks that are widely used to create graphical applications on operating-system platforms. The broad adoption of reusable GUI frameworks has yielded many productivity and quality benefits for business and desktop applications.

Application developers in more complex domains, such as telecom, financial services, process manufacturing, and aerospace, traditionally lacked reusable COTS frameworks. Developers in these domains therefore built, validated, and maintained their software from scratch. Fortunately, the current generation of reusable application server frameworks (such as JBoss, BEA's WebLogic Server, Microsoft's .NET, and ACE), network service provisioning frameworks (such as Cisco's IOS and Element Management frameworks), realtime and embedded-systems development and testing frameworks (such as TimeSys's TimeStorm IDE and MathWorks' Matlab Realtime Workshop), IDE (integrated development environment) frameworks (such as Eclipse, Microsoft's Visual Studio, and Sun's NetBeans), and CAD-enabled product-data and line-management frameworks (such as EDS's Teamcenter and EMG's E-Matrix) are designed to address a broader and deeper range of domains than GUIs.

KEY CHALLENGES IN DEVELOPING AND REUSING FRAMEWORKS

Frameworks are a promising technology for instantiating proven software designs and implementations to reduce cost and improve quality of software. Developing and using frameworks *effectively*, however, can involve considerable time and energy, depending on the complexity of the domain, the maturity of existing frameworks, the availability of good documentation, the willingness of other users who can help (e.g., mailing lists and other newsgroups on the Internet), and the ability of developers to master key concepts, patterns, features, and tools associated with frameworks. When confronted with these challenges, software developers often need to perform the following activities:

- Determine if a particular framework applies to their problem domain and whether it has sufficient quality to be an effective solution.
- Evaluate whether the time spent learning a framework outweighs the time saved by reuse.
- Learn how to debug applications written using a framework.
- Identify the performance implications of integrating application logic into a framework.
- Evaluate the effort required to develop a new framework.

Leveraging Application Frameworks

This section explores each of these activities and describes specific steps to succeed with frameworks in practice.

DETERMINING FRAMEWORK APPLICABILITY AND QUALITY

Frameworks are most applicable in problem domains where there is considerable commonality in functionality and QoS requirements of solution space, yet where each solution may vary in certain respects, thereby necessitating a framework to manage points of commonality and variability. For example, Xerces provides a powerful framework for parsing and validating the conformance of XML data to a specific DTD (document type definition) or schema. Xerces also enables the construction of data from XML files to build applications, such as XML-savvy Web servers, vertical applications that use XML as their data format, and on-the-fly validation for XML editors. The key commonality handled by the Xerces framework in all these applications is the XML parsing required to build applications that can then process the XML content in different ways using different programming languages, such as C++, Java, and Perl.

When deciding whether a framework can be used for a particular application or domain include, you should consider the following:

- Ask domain experts and product architects to identify common functionality with other domains and conduct a study of available COTS frameworks to address domain-specific and domain-independent functionality during the design phase of a project.
- Conduct pilot studies that apply various COTS frameworks to develop representative prototype applications. Such pilot studies can be conducted as part of an iterative development approach—for example, the Spiral model or XP (extreme programming).

The goal here is to identify the capabilities of existing frameworks and determine the level of effort required to integrate domain- and product-specific logic with the selected framework(s).

It's important to recognize, however, that the suitability

of a framework for a particular application may not be apparent until the learning curve has flattened, which often occurs on the second and successive projects that use the framework. Since application developers can take six to nine months to become highly productive with frameworks on their own, hands-on mentoring and training courses can help developers master a new framework more quickly and effectively. Application developers can also mitigate the effects of the learning curve by prototyping and incrementally focusing on subsets of the framework that are immediately applicable to their most immediate task at hand.

Applicability is only part of the criteria for evaluating a framework, however. The other part is *quality*—how to differentiate a good framework from a bad framework. Some specific issues to consider when evaluating the quality of a framework include the following:

- Will the framework allow applications to cleanly decouple the callback logic from the rest of the software—that is, will the framework become too tightly coupled with the development, debugging, future enhancement, and maintenance of other parts of the software?
- Can applications interact with the framework via a narrow and well-defined set of interfaces and facades?⁷ Does the framework document all the APIs that applications use to interact with the framework—for example, does it define pre-conditions and post-conditions of callback methods via contracts?
- Does the framework explicitly specify the startup, shutdown, synchronization, and memory management contracts available for the clients?

EVALUATING THE ECONOMICS OF FRAMEWORKS

Although frameworks are designed as reusable software, in practice their reusability often depends on how well they model the commonalities and variabilities across application domains, such as business data processing, telecom call processing, graphical user interfaces, or real-time middleware. By leveraging the domain knowledge and prior efforts of experienced developers, frameworks

provide solutions to common problems, and ways to extend and customize existing infrastructure to create domain-specific solutions for domain-specific problems and software design challenges. Unless the effort required to learn the framework can be amortized over many projects, however, this investment may not be cost effective; it may be better to build new capabilities in-house rather than reuse existing frameworks.

Some specific steps to take when deciding whether to reuse an existing framework or build the code include:⁸

- Determine effective framework cost metrics, which measure the savings of reusing framework components versus building applications from scratch.
- Conduct cost/effort estimations, which involves accurately forecasting the cost of buying, building, or adapting a particular framework.
- Perform investment analysis and justification, which determines the benefits of applying frameworks in terms of return on investment.

Cocomo 2.0 is an example of a widely used software cost model estimator that can help to predict the effort for new software activities. The estimates from these types of models can be used as a basis for determining the savings that could be incurred by using frameworks. A challenge confronting software development organizations, however, is that many existing software cost/effort estimation methodologies are not well calibrated to handle reusable frameworks or standards-based frameworks that provide subtle advantages, such as code portability or refactoring. Additional research is therefore necessary to characterize the appropriate techno/economic criteria for selecting frameworks.

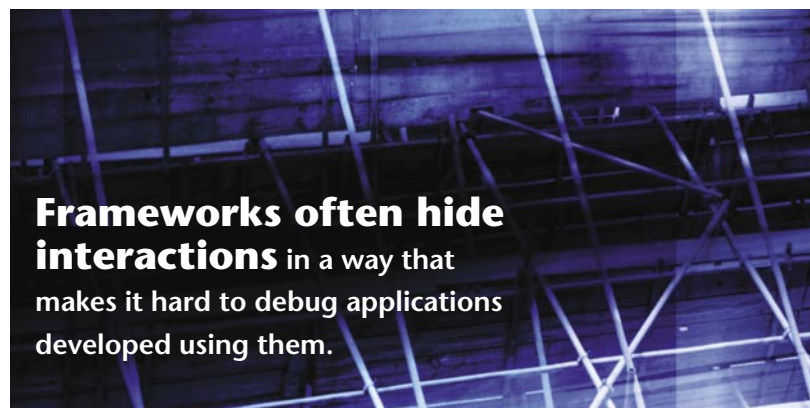
EFFECTIVE FRAMEWORK DEBUGGING TECHNIQUES

Frameworks often hide interactions in a way that makes it hard to debug applications that were developed using frameworks. As was shown in figure 1, frameworks exhibit inversion of control at runtime via callbacks to component hook methods after the occurrence of an event.

Various issues complicate the debugging of applications developed using frameworks. For example, application developers may not be intimately familiar with a framework's design and implementation, which may lead to subtle bugs caused by misinterpretations of an interface's semantics and protocols. Moreover, complex and error-prone memory management rules may be required for languages like C++ that don't support automatic garbage collection. Some frameworks also require application developers to follow subtle initialization and

termination protocols that designate the order in which objects are created or destroyed. Failure to follow these protocols correctly can cause problems that are hard to trace and debug.

Traditional techniques for debugging applications—for example, using a debugger to step through the application and verifying the state information—are often ineffective for applications built using frameworks since bugs commonly stem from faulty assumptions and misconceptions about the interactions hidden by a framework. A more effective way of debugging framework-based applications is to use tools that perform the following functions:



Frameworks often hide interactions in a way that makes it hard to debug applications developed using them.

- Track lifetimes of objects by monitoring their reference counts.
- Monitor the internal request queue lengths and buffer sizes maintained by the framework.
- Monitor the status of the network connections in distributed systems.
- Track the activities of designated threads in a thread pool.
- Trace the SQL statements issued by servers to back-end databases.
- Identify priority inversions in realtime systems.
- Track authentication and authorization activities.

Though there are many general-purpose software debugging tools, few widely used commercial tools support effective framework debugging. Projects often must develop flexible framework debugging tools that integrate the individual tool features listed above and can be configured to suit the framework being debugged. For example, debugging tools for enterprise application frameworks provide some common capabilities, such as tracking object lifetimes, network connections, threading policies, database activity, and security.

Moreover, since frameworks are often specialized for

Leveraging Application Frameworks

particular domains, good debuggers require a deep understanding of the framework's design rules to be effective. An example is OCI's Ovation, an open source tool that helps developers debug distributed applications by capturing and visually presenting: interdependencies among processes, threads, components, and objects; timing information for messages in absolute time and relative to user-defined milestones; and important epochs, such as client/server pre- and post-invoke.

These are some specific steps to reduce complexities in testing and debugging applications using frameworks:

- Perform design reviews early in the application development process to convey the types of interactions between the framework and the application logic. For example, application developers should understand the callback points in a framework and use these as starting points to help debug their applications.
- Conduct code inspections that focus on common mistakes, such as incorrectly applying memory ownership rules for preregistered components with the frameworks.
- Select good automated debugging tools, such as memory bounds checkers and code coverage instrumentation/analysis tools that help application developers identify and pinpoint common problems. Examples of these tools include Rational Purify, the open source Valgrind, and Compuware Boundschecker.
- Develop automated regression tests that exercise various framework capabilities in the context of application scenarios to get a better understanding of the strengths and weaknesses of the framework. Distributed continuous quality assurance tools, such as those shown at <http://www.dre.vanderbilt.edu/scoreboard>, can help to identify problems throughout the development cycle.

IDENTIFYING FRAMEWORK MEMORY AND PERFORMANCE OVERHEAD

Though well-written frameworks can enhance application developer productivity, they can also incur significant memory and performance overhead because of their additional generality and capabilities. Understanding these time- and space-overhead implications of frame-

works is essential for performance-sensitive applications that use frameworks along their critical paths. For example, frameworks that are used to invoke remote operations—such as CORBA (Common Object Request Broker Architecture) and Java RMI (Remote Method Invocation)—typically manage operating-system resources (such as socket connections, threads, locks, and shared memory), which can add considerable overhead if they aren't designed, implemented, or optimized properly. Common sources of time/space overhead in frameworks stem from the following factors:

Event dispatching latency—the time taken by a framework to call back application handlers when events arrive.

Synchronization latency—the time spent trying to grab and release locks along the critical path in single-threaded and multi-threaded modes of operation within a framework.

Resource management latency—the time spent trying to allocate and release resources, such as memory, and socket handles in single-threaded and multi-threaded modes of operation.

Framework functionality latency—the time spent by the thread of control within the framework for each operation it handles.

Dynamic memory overhead—which often involves the resources used to address the sources of latency just outlined. For example, a framework could cache memory allocated dynamically to reduce event dispatching latency, which in turn could increase the runtime memory of the applications that use the framework.

Static memory overhead—the amount of additional disk space that an application needs when using a framework—for example, as a result of additional framework code that is linked into an application, even though the application may not necessarily use it.

Specific steps to take when evaluating the performance of applications developed using a framework include the following:

- Conduct a systematic engineering analysis to determine the features and properties (such as scalability, tolerance to commonly occurring faults, and predictability)

required from a framework. Frameworks often perform well when a limited set of their features is used, but will perform poorly when many features (or a certain combination of features) are used.

- Develop test cases to empirically evaluate the overhead associated with every feature and combination of features. Applications in different domains may require different types of data. For example, realtime applications may require predictable low latency, whereas scientific visualization applications may require high throughput. The test cases should evaluate the required characteristics.
- Locate third-party performance benchmarks and analysis to compare with the data collected. Techniques for developing benchmarks, including regression benchmarking, are available as good reference material to develop framework benchmarking testbeds.⁹

EVALUATING THE EFFORT TO DEVELOP A NEW FRAMEWORK

Despite the depth and breadth of existing COTS frameworks, developers can still encounter situations where no existing frameworks are applicable for their domains or product needs. For example, the event loop mechanisms used to provide inversion of control in existing frameworks don't always integrate seamlessly with legacy application components. Likewise, existing frameworks may not be able to meet performance requirements or may provide insufficient information via callbacks for applications operating in certain domains (particularly applications with stringent QoS requirements). Existing frameworks may also be unusable because of lack of support for a particular programming language or operating system. In these situations, software teams may need to develop their own frameworks to accommodate the requirements in their domain.

Given how hard developing software is in general, it should be no surprise that developing high-quality, extensible,⁵ and reusable frameworks is even harder.⁶ A key challenge of designing frameworks is to decompose the framework's capabilities into a set of reusable classes, while simultaneously anticipating future uses and changes. Some specific issues that should be addressed when developing a new framework include:

- Determining which classes should be fixed, thus defining the stable shape and usage characteristics of the framework. If key interfaces in a framework aren't stable, users may have difficulty understanding and applying the framework effectively and efficiently because there will be too many degrees of freedom.

- Determining which classes should be extensible—for example, by subclassing or template instantiation—to support adaptation necessary to use the framework for new applications. If a framework can't be extended, then users can't customize it for their needs, which makes it hard to accommodate a diverse set of applications and use cases that were not foreseen during the framework's initial design.
- Determining the right protocols for startup and shutdown sequences of operations. If the application developers cannot pick and choose the initialization and termination sequences of framework operations, the lifetimes of the application and framework can get coupled in complex ways. This can reduce flexibility significantly.
- Developing the right memory management and reentrancy rules for the framework. If the framework can be used by multiple threads, framework developers should provide mechanisms to serialize access to shared data and yet determine ways to provide increased concurrency for better performance by minimizing excessive locking.
- Determining the right set of narrow interfaces for the clients to use. Too narrow an interface can lead to restrictions and place an undue burden on the application, whereas too broad an interface can lead to confusing API usage.

The diversity of the domains in which frameworks can be applied makes defining a single universal strategy for developing frameworks difficult; hard-won experience and insights are crucial ingredients to success. In general, however, well-designed frameworks are often developed via a systematic process of identifying the commonality and variability¹⁰ of policies and mechanisms in a particular application domain. The commonality should be factored into stable reusable class interfaces. The variability should be factored into reusable classes whose implementations conform to a common interface so they can be substituted easily to meet the needs of particular applications in particular contexts.

Fortunately, there are now many documented patterns⁷ and pattern languages¹ that can help guide and accelerate the design and implementation of frameworks by enabling developers to reuse higher-level software application designs, such as publisher/subscriber architectures, micro-kernels, and brokers.¹¹ These design artifacts represent some of the key strategic aspects of complex software systems. If they are understood and applied properly via frameworks, the impact of many vexing complexities can be greatly alleviated. Even so, getting the design and implementation of a framework right

Leveraging Application Frameworks

may take a number of iterations. To get a good return on the investment needed to develop a good framework, therefore, this effort must be amortized over multiple applications and projects—otherwise, the investment may simply not be cost effective.

BENEFITS TO COME

The past decade has yielded significant progress in the development and reuse of frameworks. As a result, we now have frameworks based on open standards, such as Java and CORBA, that provide a portable and interoperable set of software artifacts, such as interoperable security, distributed resource management, and fault-tolerance services. In the future, many applications will be assembled by integrating and scripting domain-specific and common “pluggable” framework components, rather than being programmed from scratch as they are today. Key topics and domains that will benefit from the foundation work on frameworks conducted thus far include:

Distributed realtime and embedded systems. An increasing number of patterns associated with frameworks for concurrent and networked systems have been documented recently.^{12,1} A key next step is to develop frameworks for DRE (distributed realtime and embedded) systems. This will extend earlier efforts to focus on effective strategies and tactics for managing key QoS properties in DRE systems, including network bandwidth and latency, CPU speed, memory access time, and power levels. Since developing high-quality DRE systems is difficult and remains something of a “black art,” relatively few reusable patterns¹³ and frameworks¹⁴ exist for this domain today. We expect an increased focus on DRE systems in the future, however, as reusable framework technology matures, together with the development tools, techniques, and processes that enable frameworks to be applied successfully in the DRE domain.

Mobile systems. Wireless networks are becoming pervasive, and embedded devices are becoming smaller, lighter, and more capable. Thus, mobile systems will soon support many consumer communication and computing needs. Application areas for mobile systems

include ubiquitous computing, mobile agents, personal assistants, position-dependent information provision, remote medical diagnostics and teleradiology, and home and office automation.⁶ In addition, Internet services, ranging from Web browsing to online banking, will be accessed from mobile systems. Mobile systems present many challenges, such as managing low and variable bandwidth and power, adapting to frequent disruptions in connectivity and service quality, diverging protocols, and maintaining cache consistency across disconnected network nodes. We expect that experienced developers of mobile systems will capture their expertise in the form of reusable frameworks to help meet the growing demand for quality software in this area.

Adaptive QoS for COTS systems. Distributed applications, such as streaming video, Internet telephony, and large-scale interactive simulation systems, have increasingly stringent QoS. To reduce development cycle time and cost, these applications are increasingly being developed using multiple layers of COTS hardware, operating systems, and middleware components. Historically, however, it has been hard to configure COTS-based systems that can simultaneously satisfy multiple QoS properties, such as security, timeliness, and fault tolerance.¹⁵ As developers and integrators continue to master the complexities of providing end-to-end QoS guarantees, it is essential that they create adaptive and reflective frameworks to help others configure, monitor, and control COTS-based distributed systems that possess a range of interdependent QoS properties.

Despite the many benefits of frameworks, however, they are not silver bullets. In particular, they don’t absolve developers from responsibility for solving all complex concurrent and networked software analysis, design, implementation, validation, and optimization problems. Ultimately, there is no substitute for human creativity, experience, discipline, diligence, and judgment. When applied using the techniques described in this article, however, frameworks can help to alleviate many accidental and inherent complexities, thereby yielding better-quality software with less overall time and effort. Q

REFERENCES

1. Schmidt, D. C., Stal, M., Rohnert, H., and Buschmann, F. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2*. Wiley & Sons, New York: NY, 2000.
2. Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, D. *Refactoring—Improving the Design of Existing Code*. Addison-Wesley, Reading: MA, 1999.
3. Foote, B., and Yoder, J. Big Ball of Mud. In *Pattern Languages of Program Design 4*, Foote, B., Harrison, N., and Rohnert, H., eds. Addison-Wesley, Boston: MA, 2000.
4. Fayad, M., Johnson, R., and Schmidt, D. C., eds. *Implementing Application Frameworks: Object-Oriented Frameworks at Work*. Wiley & Sons, New York: NY, 1999.
5. Fayad, M., Johnson, R., and Schmidt, D. C., eds. *Building Application Frameworks: Object-Oriented Foundations of Framework Design*. Wiley & Sons, New York: NY, 1999.
6. Johnson, R. Frameworks = (patterns + components). *Communications of the ACM* 40, 10 (Oct. 1997), 39-42.
7. Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading: MA, 1995.
8. Fayad, M. E., and Hamu, D. S. Enterprise frameworks: guidelines for selection. *ACM Computing Surveys* (Mar. 2000).
9. Lockheed Martin Advanced Technology Labs, ATL QoS Home Page; <http://www.atl.external.lmco.com/projects/QoS/>.
10. Coplien, J., Hoffman, D., and Weiss, D. Commonality and variability in software engineering. *IEEE Software* 15, 6 (Nov./Dec. 1998), 37-45.
11. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. *Pattern-Oriented Software Architecture—A System of Patterns*. Wiley & Sons, New York: NY, 1996.
12. Lea, D. *Concurrent Programming in Java: Design Principles and Patterns, Second Edition*. Addison-Wesley, Boston: MA, 2000.
13. Noble J., and Weir, C. *Small Memory Software: Patterns for Systems with Limited Memory*. Addison-Wesley, Boston: MA, 2001.
14. Schmidt, D. C., and Huston, S. D. *C++ Network Programming, Volume 2: Systematic Reuse with ACE and Frameworks*. Addison-Wesley, Reading: MA, 2002.
15. Zinky, J. A., Bakken, D. E., and Schantz, R. Architectural support for quality of service for CORBA objects. *Theory and Practice of Object Systems* 3, 1 (1997), 1-20.

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

DOUGLAS C. SCHMIDT, Ph.D (d.schmidt@vanderbilt.edu), is a professor in the electrical engineering and computer science department at Vanderbilt University and a senior member of the Institute for Software Integrated Systems.

He has published more than 250 technical papers and books covering patterns, optimization techniques, and empirical analyses of software frameworks that facilitate the development of DRE (distributed realtime and embedded) middleware running over high-speed networks and embedded system interconnects. Schmidt has served as a deputy office director and a program manager at DARPA (Defense Advanced Research Projects Agency), where he led the national R&D effort on middleware for DRE systems. Schmidt has more than 15 years of experience leading the development of ACE (Adaptive Communication Environment) and TAO, which are widely used, open source DRE middleware frameworks.

ANIRUDDHA S. GOKHALE (a.gokhale@vanderbilt.edu) is an assistant professor in the electrical engineering and computer science department and a research scientist at the Institute for Software Integrated Systems, both at Vanderbilt University. His research interests are in realtime component middleware optimizations, model-driven software synthesis applied to component middleware-based applications, and distributed resource management. He is leading DARPA (Defense Advanced Research Projects Agency) projects that involve modeling and middleware solutions and distributed dynamic resource management. In addition, he is heading the R&D efforts on an open source model-driven middleware framework called CoSMIC.

Gokhale was previously with Bell Laboratories, Lucent Technologies. He received his B.E. in computer engineering from the University of Pune, India; M.S. in computer science from Arizona State University; and D.Sc. in computer science from Washington University.

BALACHANDRAN NATARAJAN (bala@cs.wust.edu) is a senior staff engineer at the Institute for Software Integrated Systems and a Ph.D. student in electrical engineering and computer science at Vanderbilt University. His research focuses on applying patterns, optimization principles, and frameworks to build high-performance, dependable, and realtime distributed object computing systems. Natarajan received his M.S. in computer science at Washington University.

© 2004 ACM 1542-7730/04/0700 \$5.00