Frameworks: Why They Are Important and How to Apply Them Effectively

Douglas C. Schmidt, Aniruddha Gokhale, and Balachandran Natarajan
{schmidt,gokhale,bala}@dre.vanderbilt.edu
Department of Electrical Engineering and Computer Science
Vanderbilt University
Nashville, TN 37203

1 Introduction

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 quality of service (QoS) requirements, (3) **portable** to reduce the effort required to support applications on heterogeneous OS 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. It is hard to achieve these qualities, however, when:

- **Core concepts and software artifacts are continually rediscovered and reinvented**, *i.e.*, 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 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].
- **Software is developed monolithically**, *i.e.*, 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]. While monolithic software may sometimes be appropriate in short-lived, "throw away" prototypes [3] 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 of the software, thereby enhancing software extensibility, flexibility, and portability via

- **Design reuse**, *e.g.*, by guiding application developers through the steps necessary to ensure successful creation and deployment of complex software
- **Implementation reuse**, *e.g.*, by amortizing software lifecycle costs and leveraging previous development and optimization effort and
- **Validation reuse**, *e.g.*, 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 commercial-off-the-shelf (COTS) products they often become more affordable.

While frameworks can be a very 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 then explores key challenges that arise when developing and reusing frameworks, and describes specific steps to address these challenges.

2 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 char-

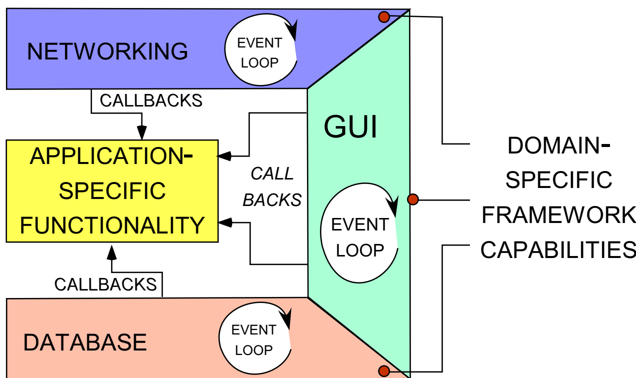

Figure 1: Relationships Between Framework Artifacts

acteristics of frameworks that help them achieve the qualities outlined at the beginning of this article. We describe each of these characteristics below:

• **A framework exhibits "inversion of control" at run time via callbacks.** These callbacks invoke the *hook methods* of application-defined components after the occurrence of 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 pre-registered 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 (1) capturing the common abstractions of an application domain – both their structure and behaviors – while (2) 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 run time 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 standalone 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 graphical user interfaces (GUIs). Java Foundation Classes (JFC), Microsoft Foundation Classes (MFC), and Qt are contemporary GUI frameworks that are widely used to create graphical applications on OS 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 historically 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), real-time and embedded systems development and testing frameworks (such as TimeSys's TimeStorm IDE and MathWorks's Matlab Real-time Workshop), integrated development environment (IDE) 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.

3 Key Challenges in Developing and Reusing Frameworks

Although 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* can involve considerable time and energy, depending on (1) the complexity of the domain, (2) the maturity of existing frameworks, (3) the availability of good documentation, (4) the willingness of other users who can help

(*e.g.*, mailing lists and other news groups on the Internet), and (5) the ability of developers to master key concepts, patterns, features and tools associated with frameworks. When confronted with these challenges, software developers often need 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.

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

3.1 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 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 document type definition (DTD) 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.

Some specific steps to take when deciding whether a framework can be used for a particular application or domain include:

- Having domain experts and product architects 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, *e.g.*, the Spiral model or eXtreme Programming (XP).

The goal of these steps 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, 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 6-9 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*, *i.e.*, how to identify 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, *i.e.*, 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 API's that are used by applications to interact with the framework, *e.g.*, 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?

3.2 Evaluating the Economics of Frameworks

Although frameworks as defined in Section 2 are designed as reusable software, in practice their (re)usability 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 provide 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 and 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]:

- **Determining effective framework cost metrics**, which measure the savings of reusing framework components vs. building applications from scratch.
- **Conducting cost/effort estimations**, which is the activity of accurately forecasting the cost of buying, building, or adapting a particular framework.
- **Performing 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 of 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.

3.3 Effective Framework Debugging Techniques

Frameworks often hide interactions in a way that makes debugging applications developed using frameworks hard. As shown in Figure 1, frameworks exhibit inversion of control at run time via callbacks to component hook methods after the occurrence of an event, such as a mouse click or data arriving on a network connection. When an event occurs, among other things the framework calls back to a virtual hook method in a pre-registered application component, which then performs application-defined processing in response to the event. The implementation of the hook methods within the components decouple the application or the business logic from the reusable framework software, which allows each to change independently as long as the interface signature and interaction protocols are not modified.

There are, however, various issues that 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 yield problems that are hard to trace and debug.

Traditional techniques used for debugging applications, *e.g.*, using a debugger to step through the application and verifying the state information, is 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 to debugging framework-based applications is to use the following tools that:

1. Track lifetimes of objects by monitoring their reference counts.
2. Monitor the internal request queue lengths and buffer sizes maintained by the framework.
3. Monitor the status of the network connections in distributed systems.
4. Track the activities of designated threads in a thread pool.
5. Trace the SQL statements issued by servers to backend databases.
6. Identify priority inversions in real-time systems.
7. Track authentication and authorization activities.

Though there are many general-purpose software debugging tools, there are few widely-used commercial tools that support effective framework debugging. It is often necessary for projects to 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 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 particular domains, good debuggers require a deep understanding of the framework's design rules to be effective. An example of such a tool is OCI's OVATION, which is an open-source tool that helps developers debug distributed applications by capturing and visually presenting (1) interdependencies between processes, threads, components, and objects, (2) timing information for messages in absolute time and relative to user-defined milestones, and (3) important epochs, such as client/server pre- and post-invoke.

Some specific steps that can be taken to reduce complexities in testing and debugging applications using frameworks include:

1. **Performing design reviews** early in the application development process to convey the type 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.

2. **Conducting code inspections** that focus on common mistakes, such as incorrectly applying memory ownership rules for pre-registered components with the frameworks.
3. **Selecting good automated debugging tools**, such as memory bounds checkers and code coverage instrumentation/analysis tools that help application developers identify and pinpoint common problems such as Rational Purify, Valgrind, and Compuware Boundschecker.
4. **Developing 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 `www.dre.vanderbilt.edu/scoreboard`, can help to identify problems throughout the development cycle.

3.4 Identifying Framework Memory and Performance Overhead

Though well-written frameworks can enhance application developer productivity, they can also incur significant memory and performance overhead due to their additional generality and capabilities. Understanding these time and space overhead implications of frameworks is essential for performance-sensitive applications that use frameworks along their critical path. For example, frameworks that are used to invoke remote operations (such as CORBA and Java RMI) typically manage OS resources (such as socket connections, threads, locks, and shared memory), which can add considerable overhead to 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**, which is the time taken by a framework to callback application handlers when events arrive.
- **Synchronization latency**, which is the duration of time spent trying to grab and release locks along the critical path in single-threaded and multi-threaded mode of operation within a framework.
- **Resource management latency**, which is the duration of time spent trying to allocate and release resources, such as memory, and socket handles in single-threaded and multi-threaded mode of operation.
- **Framework functionality latency**, which is 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 outlined above. 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**, which is the amount of additional disk space that an application uses when using a framework, *e.g.*, due to additional framework code that is linked into an application, even though the application may not necessarily use it.

Some specific steps to take when evaluating the performance of applications developed a framework include:

- Conducting a systematic engineering analysis to determine the features and properties (such as scalability, tolerance to commonly occuring faults, and predictability) required from a framework. Frameworks often perform well when a limited set of their features are used, but will perform poorly when many features (or a certain combination of features) are used.
- Developing 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, real-time applications may require predictable low latency, whereas scientific visualization applications may require high throughput. The test cases should evaluate the required characteristics.
- Locating third-party performance benchmarks and analysis to compare with the data collected. Techniques for developing benchmarks including regression benchmarking are available [9] as good reference material to develop framework benchmarking testbeds.

3.5 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 domain or product needs. For instance, 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 due to 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 it is to develop software in general, it should be no surprise that developing high quality, extensible, 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, it may be hard for users to understand and apply the framework effectively and efficiently because there will be too many degrees of freedom.
- Which classes should be extensible, *e.g.*, 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, which can reduce flexibility significantly.
- Developing right memory management and re-entrancy 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.
- The right set of narrow interfaces that can be used by the clients. Too narrow an interface can lead to restrictions and place 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 make it hard to define a single universal strategy for developing frameworks, *i.e.*, 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 variability [10] 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 patterns [7] and pattern languages [1] 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, however, it may take a number of iterations to get the design and implementation of a framework right. 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.

4 Concluding Remarks

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 like they are today. Key topics and domains that will benefit from the foundational work on frameworks conducted thus far include:

- **Distributed real-time 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 distributed real-time and embedded (DRE) systems, which extends 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 hard and remains of a "black art," relatively few reusable patterns [13] and frameworks [14], 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 become 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. In addition, Internet services, ranging from Web browsing to on-line 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 judgement. 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.

References

[1] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2*. New York: Wiley & Sons, 2000.

[2] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring - Improving the Design of Existing Code*. Reading, Massachusetts: Addison-Wesley, 1999.

[3] B. Foote and J. Yoder, "Big Ball of Mud," in *Pattern Languages of Program Design 4* (B. Foote, N. Harrison, and H. Rohnert, eds.), Boston: Addison-Wesley, 2000.

[4] M. Fayad, R. Johnson, and D. C. Schmidt, eds., *Implementing Application Frameworks: Object-Oriented Frameworks at Work*. New York: Wiley & Sons, 1999.

[5] M. Fayad, R. Johnson, and D. C. Schmidt, eds., *Building Application Frameworks: Object-Oriented Foundations of Framework Design*. New York: Wiley & Sons, 1999.

[6] R. Johnson, "Frameworks = Patterns + Components," *Communications of the ACM*, vol. 40, Oct. 1997.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[8] M. E. Fayad and D. S. Hamu, "Enterprise Frameworks: Guidelines for Selection," *ACM, COmputing Surveys*, Mar. 2000.

[9] L. M. A. T. Labs, "ATL QoS Home Page." `www.atl.external.lmco.com/projects/QoS/`, 2002.

[10] J. Coplien, D. Hoffman, and D. Weiss, "Commonality and Variability in Software Engineering," *IEEE Software*, vol. 15, November/December 1998.

[11] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture—A System of Patterns*. New York: Wiley & Sons, 1996.

[12] D. Lea, *Concurrent Programming in Java: Design Principles and Patterns, Second Edition*. Boston: Addison-Wesley, 2000.

[13] J. Noble and C. Weir, *Small Memory Software: Patterns for Systems with Limited Memory*. Boston: Addison-Wesley, 2001.

[14] D. C. Schmidt and S. D. Huston, *C++ Network Programming, Volume 2: Systematic Reuse with ACE and Frameworks*. Reading, Massachusetts: Addison-Wesley, 2002.

[15] J. A. Zinky, D. E. Bakken, and R. Schantz, "Architectural Support for Quality of Service for CORBA Objects," *Theory and Practice of Object Systems*, vol. 3, no. 1, pp. 1–20, 1997.