Object-Oriented Application Frameworks

Computing power and network bandwidth have increased dramatically over the past decade, yet the design and implementation of complex software remain expensive and error-prone. Much of the cost and effort stems from the continuous rediscovery and reinvention of core concepts and components across the software industry. In particular, the growing heterogeneity of hardware architectures and diversity of operating system and communication platforms make it difficult to build correct, portable, efficient, and inexpensive applications from scratch.

Mohamed E. Fayad and Douglas C. Schmidt, *Guest Editors*

Object-oriented application frameworks are a promising technology for reifying proven software designs and implementations in order to reduce the cost and improve the quality of software. A framework is a reusable, "semi-complete" application that can be specialized to produce custom applications [7, 10]. In contrast to earlier OO reuse techniques based on class libraries, frameworks are targeted for particular business units (such as data processing or cellular communications) and application domains (such as user interfaces or real-time avionics) [11]. Frameworks like MacApp, ET++, Interviews, ACE, Microsoft's MFC and DCOM, JavaSoft's RMI, and implementations of OMG's CORBA play an increasingly important role in contemporary software development.

The primary benefits of OO application frameworks stem from the modularity, reusability, extensibility, and inversion of control they provide to developers.

Frameworks enhance modularity by encapsulating volatile implementation details behind stable interfaces. Framework modularity helps improve software quality by localizing the impact of design and implementation changes, which reduces the effort required

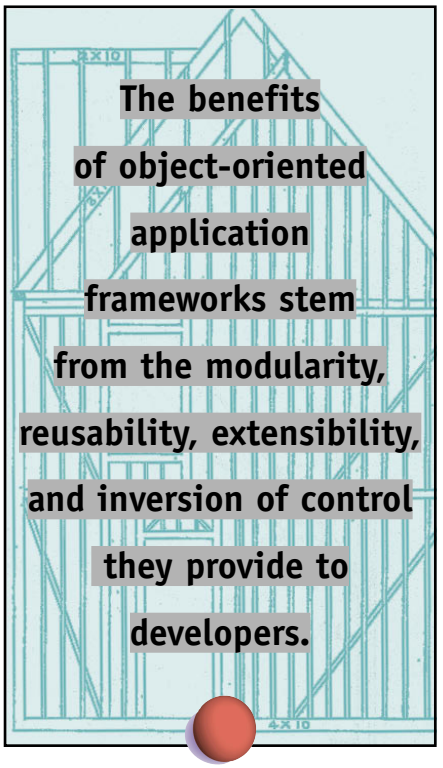
VICTOR SADOWSKI

to understand and maintain existing software.

The stable interfaces provided by frameworks enhance reusability by defining generic components that can be reapplied to create new applications. Framework reusability leverages the domain knowledge and prior effort of experienced developers in order to avoid re-creating and revalidating common solutions to recurring application requirements and software design challenges. Reuse of framework components can yield substantial improvements in programmer productivity, as well as enhancing the quality, performance, reliability and interoperability of software.

A framework enhances extensibility by providing explicit hook methods [8] that allow applications to extend its stable interfaces. Hook methods systematically decouple the stable interfaces and behaviors of an application domain from the variations required by instantiations of an application in a particular context. Framework extensibility is essential to ensure timely customization of new application services and features.

The run-time architecture of a framework is characterized by an inversion of control. This architecture enables canonical application processing steps to be customized by event handler objects that are invoked via the framework's reactive dispatching mechanism. When events occur, the framework's dispatcher reacts by invoking hook methods on pre-registered handler objects, which perform application-specific processing on the events. Inversion of control allows the framework (rather than each application) to determine which set of application-specific methods to invoke in response to external events (such as window messages arriving from end users or packets arriving on particular communication ports).



Overview of Widely Used Frameworks

Developers in certain domains have successfully applied OO application frameworks for many years. Early object-oriented frameworks (such as MacApp and Interviews) originated in the domain of graphical user interfaces. The Microsoft Foundation Classes (MFC) is a contemporary GUI framework that has become the de facto industry standard for creating graphical applications on PC platforms. Although MFC has limitations (such as lack of portability to non-PC platforms), its widespread adoption demonstrates the productivity benefits of reusing common frameworks to develop graphical business applications.

Application developers in more complex domains (such as telecommunications, distributed medical imaging, and real-time avionics) have traditionally lacked standard "off-the-shelf" frameworks. As a result, developers in these domains largely build, validate, and maintain software systems from scratch. In an era of deregulation and global competition, however, it has become prohibitively expensive and extremely time-consuming to develop applications entirely in-house.

Fortunately, the next generation of OO application frameworks are targeting complex business and application domains. At the heart of this effort are Object Request Broker (ORB) frameworks, which facilitate communication between local and remote objects. ORB frameworks eliminate many tedious, error-prone, and non-portable aspects of creating and managing distributed applications and reusable service components. This enables programmers to develop and deploy complex applications rapidly and robustly, rather than wrestling endlessly with low-level infrastructure concerns.

Classifying Application Frameworks

Although the benefits and design principles underlying frameworks are largely independent of the domains to which they are applied, we've found it useful to classify frameworks by their scope, as follows:

System infrastructure frameworks simplify the development of portable and efficient system infrastructure such as operating system [2] and communication frameworks [9], and frameworks for user interfaces and language processing tools. System infrastructure frameworks are primarily used internally within a software organization and are not sold to customers directly.

Middleware integration frameworks are commonly used to integrate distributed applications and components. Middleware integration frameworks are

designed to enhance the ability of software developers to modularize, reuse, and extend their software infrastructure to work seamlessly in a distributed environment. Middleware integration frameworks represent a thriving market, and are rapidly becoming commodities. Common examples include ORB frameworks, message-oriented middleware, and transactional databases.

Enterprise application frameworks address broad application domains (such as telecommunications, avionics, manufacturing, and financial engineering [1, 10, 11]) and are the cornerstone of enterprise business activities [3]. Relative to system infrastructure and middleware integration frameworks, enterprise frameworks are expensive to develop and/or purchase. However, enterprise frameworks can provide a substantial return on investment since they support the development of end-user applications and products directly. In contrast, system infrastructure and middleware integration frameworks focus largely on internal software development concerns. Although these frameworks are essential to create high-quality software rapidly, they typically do not generate substantial revenue for large enterprises. As a result, it is often more cost-effective to buy system infrastructure and middleware integration frameworks rather than build them in-house [3, 4, 11].

Regardless of their scope, frameworks can also be classified by the techniques used to extend them, which range along a continuum from white-box frameworks to black-box frameworks. White-box frameworks rely heavily on OO language features like inheritance and dynamic binding in order to achieve extensibility. Existing functionality is reused and extended by (1) inheriting from framework base classes and (2) overriding pre-defined hook methods using patterns like the Template Method [5]. Black-box frameworks support extensibility by defining interfaces for components that can be plugged into the framework via object composition. Existing functionality is reused by (1) defining components that conform to a particular interface and (2) integrating these components into the framework using patterns like Strategy [5] and Functor.

White-box frameworks require application developers to have intimate knowledge of each framework's internal structure. Although white-box frameworks are widely used, they tend to produce systems that are tightly coupled to the specific details of the framework's inheritance hierarchies. In contrast, black-box frameworks are structured using object composition and delegation rather than inheritance. As a result, black-box frameworks are generally easier to use and extend than white-box frameworks. However, black-box frameworks are more difficult to develop since they require framework developers to define interfaces and hooks that anticipate a wider range of potential use cases [6].

Strengths and Weaknesses of Application Frameworks

When used in conjunction with patterns, class libraries, and components, OO application frameworks can significantly increase software quality and reduce development effort. However, a number of challenges must be addressed in order to employ frameworks effectively. Companies attempting to build or use large-scale reusable frameworks often fail unless they recognize and resolve challenges such as development effort, learning curve, integrability, maintainability, validation and defect removal, efficiency, and lack of standards [10].

- While developing complex software is difficult enough, developing high quality, extensible, and reusable frameworks for complex application domains is even harder. The skills required to produce frameworks successfully often remain locked in the heads of expert developers. One of the goals of this special section is to demystify the software process and design principles associated with developing and using frameworks.
- Learning to use an OO application framework effectively requires considerable investment of effort. For instance, it often takes 6–12 months to become highly productive with a GUI framework like MFC or MacApp, depending on the experience of the developers. Typically, hands-on mentoring and training courses are required to teach application developers how to use such a framework effectively. Unless the effort required to learn a framework can be amortized over many projects, this investment may not be cost-effective. Moreover, the suitability of a framework for a particular application may not be apparent until the learning curve has flattened.
- Application development will be increasingly based on the integration of multiple frameworks (e.g., GUIs, communication systems, databases) together with class libraries, legacy systems, and existing components. However, many earlier generation frameworks were designed for internal extension rather than for integration with other frameworks developed externally. Integration problems arise at several levels of abstraction, ranging from documentation issues [3, 4], to the concurrency/distribution architecture, to the

event dispatching model. For instance, while inversion of control is an essential feature of a framework, integrating frameworks with event loops that are not designed to interoperate with other frameworks is hard.

- Application requirements change frequently. Therefore, the requirements of frameworks often change as well. As frameworks evolve, the applications that use them must evolve with them.
- Framework maintenance activities include modification and adaptation of the framework. Both modification and adaptation may occur on the functional level (i.e., certain framework functionality does not fully meet developers' requirements), as well as on the non-functional level (which includes more qualitative aspects such as portability or reusability).
- Framework maintenance may take different forms, such as adding functionality, removing functionality, and generalization. A deep understanding of the framework components and their interrelationships is essential to perform this task successfully. In some cases, the application developers and/or the end-users must rely entirely on framework developers to maintain the framework.
- Although a well-designed modular framework can localize the impact of software defects, validating and debugging applications built using frameworks can be tricky for the following reasons:
- Generic components are harder to validate in the abstract. A well-designed framework component typically avoids application-specific details, which are provided via subclassing, object composition, or template parameterization. While this improves the flexibility and extensibility of the framework, it greatly complicates module testing since the components cannot be validated in isolation from their specific instantiations. It is usually hard to distinguish bugs in the framework from bugs in the application code. As with any software development, bugs are introduced into a framework from many possible sources, such as failure to understand the requirements, overly coupled design, or an incorrect implementation. When customizing the components in a framework to a particular application, the number of possible error sources will increase.
- Inversion of control and lack of explicit control flow. Applications written with frameworks can be hard to debug since the framework's "inverted" flow of control oscillates between the application-independent framework infrastructure and the application-specific method callbacks. This increases the difficulty of "single-stepping" through the run-time behavior of a framework within a debugger since the control flow of the application is driven implicitly by callbacks and developers may not understand or have access to the framework code. This is similar to the problems encountered trying to debug a compiler lexical analyzer and parser written with LEX and YACC. In these applications, debugging is straightforward when the

Future Trends

Over the next several years, we expect the following framework-related topics will receive considerable attention from researchers and developers:

Reducing framework development effort. Traditionally, reusable frameworks have been developed by generalizing from existing systems and applications. Unfortunately, this incremental process of organic development is often slow and unpredictable since core framework design principles and patterns must be discovered from the "bottom-up." However, since many good framework examples now exist, we expect that the next generation of developers will leverage this collective knowledge to conceive, design, and implement higher-quality frameworks more rapidly [11].

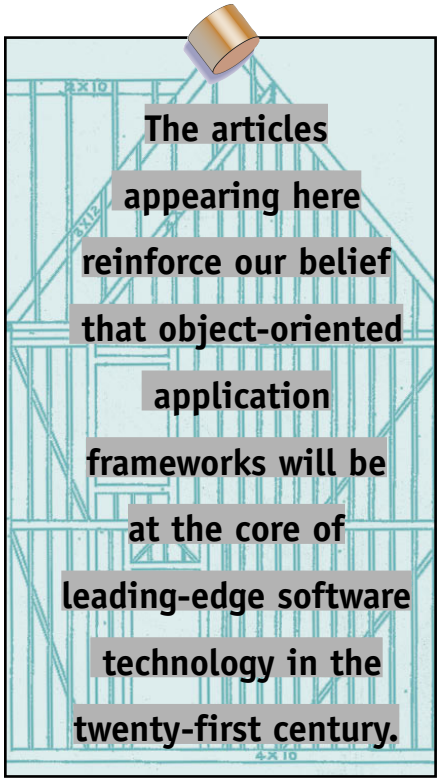
Greater focus on domain-specific enterprise frameworks. Existing frameworks have focused largely on system infrastructure and middleware integration domains (such as user interfaces [5, 8] and OS/communication systems [2, 6, 9, 11]). In contrast, there are relatively few widely documented examples of enterprise frameworks for key business domains such as manufacturing, banking, insurance, and medical systems [4, 11]. As more experience is gained developing frameworks for these business domains, however, we expect that the collective knowledge of frameworks will be expanded to cover an increasingly wide range of domain-specific topics and an increasing number of enterprise application frameworks will be produced [3, 4]. As a result, benefits of frameworks will become more immediate to application programmers as well as to infrastructure developers.

Black-box frameworks. Many framework experts [7, 10] favor black-box frameworks over white-box frameworks since black-box frameworks emphasize dynamic object relationships (via patterns like Bridge and Strategy [5]) rather than static class relationships. Thus, it is easier to extend and reconfigure black-box frameworks dynamically. As developers become more familiar with techniques and patterns for factoring out common interfaces and components, we expect that an increasing percentage of black-box frameworks will be produced.

Framework documentation. Accurate and comprehensible documentation is crucial to the success of large-scale frameworks.

thread of control is in the user-defined action routines. Once the thread of control returns to the generated DFA skeleton, however, it is hard to trace the program's logic.

Frameworks enhance extensibility by employing additional levels of indirection. For instance, dynamic binding is commonly used to allow developers to subclass and customize existing interfaces. However, the resulting generality and flexibility often reduce efficiency. For instance, in languages like C++ and Java, the use of dynamic binding makes it impractical to support Concrete Data Types (CDTs), which are often required for time-critical software. The lack of CDTs yields (1) an increase in storage layout (due to embedded pointers to virtual tables), (2) performance degradation (due to the additional overhead of invoking a dynamically bound method and the inability to inline small methods), and (3) a lack of flexibility (due to the inability to place objects in shared memory).



Currently, there are no widely accepted standards for designing, implementing, documenting, and adapting frameworks. Moreover, emerging industry standard frameworks (such as CORBA, DCOM, and Java RMI) currently lack the semantics, features and interoperability to be truly effective across multiple application domains. Often, vendors use industry standards to sell proprietary software under the guise of open systems. Therefore, it is essential for companies and developers to work with standards organizations and middleware vendors to ensure that the emerging specifications support true interoperability and define features that meet their software needs.

The Articles

The articles in this section describe how OO application frameworks provide a powerful vehicle for reuse, as well as a way to capture the essence of successful patterns, architectures, components, policies, services, and programming mechanisms. We begin with an overview of the topic by Ralph Johnson. "Frameworks = (Components + Patterns)" compares and contrasts frameworks with other object-oriented reuse techniques—patterns and components.

"An Adaptive Framework for Developing Multimedia Software Components" by Posnak, Lavender,

However, documenting frameworks is a costly activity and contemporary tools often focus on low-level method-oriented documentation, which fails to capture the strategic roles and collaborations among framework components. We expect that the advent of tools for reverse-engineering the structure of classes and objects in complex frameworks will help to improve the accuracy and utility of framework documentation. Likewise, we expect to see an increase in the current trend [4, 6, 9] of using design patterns to provide higher-level descriptions of frameworks.

Processes for managing framework development. Frameworks are inherently abstract since they generalize from a solution to a particular application challenge to provide a family of solutions. This level of abstraction makes it difficult to engineer their quality and manage their production. Therefore, it is essential to capture and articulate development processes that can ensure the successful development and use of frameworks. We believe that extensive prototyping and phased introduction of framework technology into organizations is crucial to reducing risk and helping to ensure successful adoption [4, 10].

Framework economics. The economics of developing frameworks include activities [4, 10] such as the following:

- *Determining effective framework cost metrics,* which measure the savings of reusing framework components vs. building applications from scratch;
- *Cost estimation*, which involves accurately forecasting the cost of buying, building, or adapting a particular framework; and
- *Investment analysis and justification,* which determines the benefits of applying frameworks in terms of return on investment.

We expect that the focus on framework economics will help to bridge the gap among the technical, managerial, and financial aspects of making, buying, or adapting frameworks [4].

Framework standards. In order to develop, document, integrate, and adapt long-lived application frameworks, standards are a must. As application frameworks become more complex and more widely accepted, standards become invaluable and increasingly essential. Standards assure consistency, and form a base to justify the framework cost and protect the investment. We believe that several standards will emerge, such as framework development, framework adaptation, framework interoperability and integration standards [10].

and Vin describes a framework that simplifies the development of dynamically adaptive multimedia software components by promoting the reuse of code, design patterns, and domain expertise. Hans Albrecht Schmid's article "Systematic Framework Design by Generalization" presents a systematic method for designing frameworks based on identifying "hot spots," which capture key sources of variation in an application domain.

"Framework Development for Large Systems" by Bäumer et al. draws upon the authors' experience developing large-scale industrial banking projects to present concepts and techniques for domain partitioning, framework layering, and framework construction.

Demeyer, Meijler, Nierstrasz, and Steyaert also focus on hot spots in their article "Design Guidelines for 'Tailorable' Frameworks," which presents design guidelines for developing frameworks for open systems. "The Framework Life Span" by Brugali, Menga, and Aarsten highlights the relationships between application frameworks, patterns, and pattern languages in the domain of manufacturing systems. "From Custom Applications to Domain-Specific Frameworks" by Codenie et al. discusses solutions to common framework development challenges such as avoiding the proliferation of versions, estimating effort and alleviating the tendency toward architectural drift.

Adele Goldberg, Steven Abell, and David Leibs describe LearningWorks—a framework for exploring ideas about computing and software system construction—in a succinct contribution to this section. Another short article, "SEMATECH's Experiences with the CIM Framework," by Doscher and Hodges, describes the structure of a framework for computer-integrated manufacturing of semiconductors.

We conclude the section with a brief consideration of lessons learned from our varied experiences applying frameworks to real-world business situations.

The articles appearing here reinforce our belief that object-oriented application frameworks will be at the core of leading-edge software technology in the twenty-first century. The extensive focus on application frameworks in the object-oriented community offers software developers an important vehicle for reuse and a means to capture the essence of successful patterns, architectures, components, and programming mechanisms.

It is significant that frameworks are becoming mainstream and that developers at all levels are increasingly adopting and succeeding with framework technologies. However, OO application frameworks are ultimately only as good as the people who build and use them. Creating robust, efficient, and reusable application frameworks requires development teams with a wide range of skills. We need expert analysts and designers who have mastered patterns, software architectures, and protocols in order to alleviate the inherent and accidental complexities of complex software. Likewise, we need expert middleware developers who can implement these patterns, architectures, and protocols within reusable frameworks. In addition, we need application programmers who have the motivation, skills, and training to learn how to use these frameworks effectively. We encourage you to get involved with others working on frameworks by attending conferences, participating in online mailing lists and newsgroups, and contributing your insights and experiences. More information can be found about this subject at http://www.cs.unr.edu/~fayad/frameworks. C

References

1. Birrer, E.T. Frameworks in the financial engineering domain: An experience report. In *Proceedings of ECOOP '93 Proceedings, Lecture Notes in Computer Science nr. 707*, Springer-Verlag, 1993.
2. Campbell, R.H. and Islam, N. A technique for documenting the framework of an object-oriented system. *Computing Systems 6*, 4 (Fall 1993).
3. Fayad, M.E. and Hamu, D.S. Object-oriented enterprise frameworks: Make vs. buy decisions and guidelines for selection. *Commun. ACM*, submitted for publication.
4. Fayad, M.E. and Hamu, D.S. *Object-Oriented Enterprise Frameworks*. Wiley, NY, 1997, to appear.
5. Gamma, E, Helm, R., Johnson, R. and Vlissides, J. *Design Patterns: Elements of Reusable Software Architecture*. Addison-Wesley, Reading, Mass., 1995.
6. Hueni, H., Johnson, R., and Engel, R. A framework for network protocol software. In *Proceedings of OOPSLA'95*, (Austin, Texas, Oct. 1995).
7. Johnson, R.E. and Foote, B. Designing reusable classes. *J. Object-Oriented Programming 1*, 5 (June/July 1988), 22–35.
8. Pree, W. *Design Patterns for Object-Oriented Software Development*. Addison-Wesley, Reading, Mass. 1994.
9. Schmidt, D.C. Applying design patterns and frameworks to develop object-oriented communication software. In P. Salus, Ed., *Handbook of Programming Languages, Volume I*, MacMillan Computer Publishing, 1997.
10. Fayad, M.E., Schmidt, D.C., and Johnson, R.E. *Object-Oriented Application Frameworks: Problems and Perspectives*. Wiley, NY, 1997, to appear.
11. Fayad, M.E., Schmidt, D.C., and Johnson, R.E. *Object-Oriented Application Frameworks: Implementation and Experience*. Wiley, NY, 1997, to appear.

Mohamed Fayad (fayad@cs.unr.edu) is an associate professor in the College of Engineering at the University of Nevada-Reno.
Douglas C. Schmidt (schmidt@cs.wustl.edu) is an assistant professor in the Department of Computer Science at Washington University in St. Louis, Missouri.