Lessons learned: building reusable OO frameworks for distributed software.

by Douglas C Schmidt and Mohamed E Fayad

Success in developing and deploying reusable object-oriented software components and frameworks depends on many factors, many of which are not technical in nature, and can be optimized through processes based on iteration and incremental growth. Being aware of the benefits and limitations of standards, practices and available resources also helps the software developer to be more effective. Cooperation between end-users and developers will further help improve the quality of object-oriented frameworks and components and move these into the computing mainstream.

Developing complex distributed applications can be an expensive and error-prone process. As a result, contemporary organizations are increasingly faced with a "distributed software crisis" - computing hardware and networks get smaller, faster, and cheaper, yet distributed software gets larger, slower, and more expensive to develop and maintain. The challenges of building distributed software stem from inherent and accidental complexities [3] associated with distributed systems:

* Inherent complexity stems from the fundamental challenges of developing distributed software. Chief among these is detecting and recovering from network and host failures, minimizing the impact of communication latency, and determining an optimal partitioning of service components and workload onto processing elements throughout a network.

* Accidental complexity stems from limitations with tools and techniques used to develop distributed software. A common source of accidental complexity is the widespread use of algorithmic decomposition [1] (also known as functional design), which results in non-extensible and non-reusable software designs and implementations.

The lack of extensibility and reuse is particularly problematic for complex distributed software. Extensibility is essential to ensure timely modification and enhancement of services and features. Reuse is essential to leverage the domain knowledge of expert developers to avoid redeveloping and revalidating available common solutions to recurring requirements and software challenges. Object-oriented frameworks are promising technologies for increasing the extensibility and reuse of distributed software.

Over the past decade, we have worked with many companies and agencies (including Motorola, U.S. Sprint, Ericsson, Siemens, Bellcore, Kodak, McDonnell Douglas, and the U.S. Naval Research Laboratory) building reusable OO communication software frameworks and applications. In these projects, we've applied a range of OO middleware frameworks including OMG CORBA (an emerging industry standard for distributed object computing middleware) and the ACE framework (a widely used C++ framework that implements many strategic and tactical design patterns for concurrent communication software). Some of the important lessons we've learned from developing and deploying reusable OO communication software components and frameworks in practice are described here.

Successful reuse generally requires the presence of certain key non-technical prerequisites. Many political, economic, organizational, and psychological factors can impede the successful reuse of distributed software. We have found that reuse works best when (1) the marketplace is competitive (time-to-market is crucial, so leveraging existing software substantially reduces the entire project's development effort and cost), (2) the application domain is non-trivial (repeatedly developing complete solutions from scratch is too costly), and (3) the corporate culture is supportive of an effective reuse process (developers are rewarded for taking the time to build robust, efficient, and reusable software components).

When these prerequisites do not exist, we have found that developers often fall victim to the "not-invented-here" syndrome and have a tendency to rebuild everything from scratch. Unfortunately, this situation forces them to rediscover and reinvent the core distributed software concepts and components, which is time-consuming, error-prone, and expensive.

Development processes that encourage iteration and incremental growth are essential. Expanding on the corporate culture theme, we have observed that it is crucial for top-level software managers to openly support the fact that good components, frameworks, and software architectures take time to craft and hone. If this support does not occur, we've found that many developers and project managers will take the path of least resistance and not risk their schedules and budgets by planning for reuse. Therefore, for reuse to succeed at large, organizations must have the collective vision and managerial resolve to support the incremental evolution of reusable software.

Lessons learned: building reusable OO frameworks for distributed software.

In many domains, we've observed that an 80% solution that can be evolved and optimized is preferable to trying to achieve a 100% solution that is never completed. Fred Brook's observation "Plan to throw the first one away, you will anyway" [3] applies as much today as it did 20 years ago.

Integrate framework infrastructure developers with application developers. A time-honored way of producing reusable components is to generalize from the bottom up from working systems and applications. Most of the useful components and frameworks we've encountered emerge from solving real problems in domains like telecommunications, medical imaging, avionics, and transaction processing. Therefore, we advise resisting the temptation to create "component teams" that build reusable frameworks in complete isolation from application teams. We have learned the hard way that without intimate feedback from application developers, the software artifacts produced by a component team won't solve real problems and will not be widely reused.

Industry "standards" are not panaceas. Expecting emerging industry middleware standards (like CORBA, DCOM, or Java RMI) to eliminate distributed software complexity today is very risky. For instance, although lower-level middleware implementations (such as ORBs and message-oriented middleware) are reaching maturity, the semantics of higher-level middleware services (such as the CORBA's Common Object Services and Common Facilities) are still vague, under-specific, and non-interoperable. However, despite the fact that higher-level middleware frameworks aren't quite suited to meet demanding real-time performance and reliability requirements in certain domains, we expect that over the next two years we'll see the emergence of middleware products that support such features [5].

Beware of simple(-minded) solutions to complex software problems. While developing high-quality reusable software is hard enough, developing high-quality extensible and reusable distributed middleware framework software is even harder. Not surprisingly, many companies attempting to build reusable middleware frameworks fail - often with enormous losses of money, time, and market share. We've noticed that the fear of failure often encourages companies to pin their hopes on silver bullets intended to slay the demons of distributed software complexity by using CASE tools or point-and-click wizards.

Unfortunately, simple solutions to complex problems that sound too good to be true usually are. For example, translating code entirely from high-level specifications or using trendy OO design methodologies and programming languages is no guarantee of success. In our experience, there's simply no substitute for skilled software developers, which leads to the following "lesson learned."

Respect and reward quality developers. Ultimately, reusable components are only as good as the people who build and use them. In our experience, cultivating high-quality software developers is time consuming and expensive. Ironically, many companies treat their developers as interchangeable, "unskilled labor" who can be replaced easily. We expect that over time, companies who respect and reward their high-quality software developers will increasingly outperform those who don't.

Recognize and understand:

* The interoperability and the unawareness of existing repositories and applications.

* Most of the existing ORBs do not support (or provide poor support) for dynamic invocation.

* CORBA does not comply well with type-oriented paradigms.

Beware of the integration problems. A navigator and configurator must exist to ease integration problems.

Be wary of the existence of rich set of tools, environments, and large investments in legacy systems. Please say no to reverse engineering, reengineering, or forward engineering and use wrapper or object shell [4] approaches instead.

Developing reusable OO middleware components and frameworks is not a silver bullet. Software is inherently abstract, which makes it hard to engineer its quality and to manage its production. The good news, however, is that OO component and framework technologies are becoming mainstream. Developers and users are increasingly adopting and succeeding with object-oriented design and programming.

On the other hand, the bad news is that (1) existing OO components and frameworks are largely focused on only a few areas (GUIs), (2) the skills required to successfully produce distributed middleware remain a "black art," and (3) existing industry standards still lack the semantics, features, and interoperability to be truly effective throughout the distributed software domain. Too often, vendors use industry standards to sell proprietary software under the guise of open systems. Therefore, it is essential for end users to work with standards organizations and middleware vendors to ensure the emerging specifications support true interoperability and define features that meet distributed software requirements.

Lessons learned: building reusable OO frameworks for distributed software.

To support the standardization effort, it is crucial for us to capture and document the patterns that underlie the successful distributed software components and frameworks that do exist. Likewise, we need to reify these patterns to guide the creation of standard frameworks and components for the distributed domain. We are optimistic that the next generation of OO frameworks and components will be a substantial improvement over those we've worked with in the past.

REFERENCES

1. Booch, G. Object-Oriented Analysis and Design. Benjamin-Cummings, 1993.

2. Brooks, F.P. No silver bullet: Essence and accidents of software engineering. IEEE Comput. 20, 4 (Apr. 1987), 10-19.

3. Brooks, F.P. The Mythical Man-Month. Addison-Wesley, Reading, Mass., 1975.

4. Fayad, M.E., Tsai, W.T and Fulghum, M.L. Transition to object-oriented software development. Commun. ACM 39, 2 (Feb. 1996).

5. Gokhale, A., Schmidt, D.C., Harrison, T., and Parulkar, G. Towards real-time CORBA. IEEE Communications Magazine 14, 2 (Feb. 1997).

MOHAMED E. 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.