

White Paper

# The Component Buyer's Guide

How to Cost Justify and  
Invest in Reusable  
Software Components

[www.roguewave.com](http://www.roguewave.com)



**Rogue Wave**  
SOFTWARE

# **The Component Buyer's Guide**

How to Cost Justify and  
Invest in Reusable  
Software Components

# Table of contents

---

## **Introduction 1**

### **Benefits of employing reusable components in software architectures 1**

- Reduced development time 2
- Increased developer productivity 2
- Application flexibility 2
- Application reliability 3
- Component reusability 3
- High return on investment 3
- Risk mitigation 3

### **Calculating your return on investment (ROI) 4**

- Building a component in house vs. purchasing off the shelf 4
- Lowering maintenance costs 5
- Saving money by saving time 6

### **Evaluating component features 7**

- Well-written code 7
- Extensible code 7
- Scalable to the enterprise 7
- Portable or platform-specific 8
- Support for internationalization 8
- Database-friendly 8
- Interoperable between languages 9
- Integrated within the object architecture 9
- Thread-safe 10

### **Evaluating a component vendor 10**

- Excellent reputation in the marketplace 10
- Architectural vision 11
- Breadth of solutions 11
- Standards-aware 11
- Non-intrusive implementation 11
- Thoroughly tested products 12
- Complete documentation 12
- Technical support 12
- Consulting, mentoring, and training 13

## **Summary 13**

## **References 14**

## Introduction

---

Today's complex, distributed computing environment is making tougher demands on your software design team every day. At the same time, you're being asked to get by with a smaller development staff and a leaner budget than ever before. Finding skilled software engineers is getting harder, and even if you can pull together a well-trained team, the cost of retaining talent is skyrocketing. As a result, you must find ways to streamline your development process while producing low-risk, adaptable, high-quality applications that will help you grow your business and maintain a competitive edge in the marketplace. Employing reusable software components in application design and implementation is a key solution to this common predicament.

This report:

- Reviews the many benefits of using components in software development
- Examines three ways to look at your return on investment
- Outlines issues to keep in mind when evaluating software components and related services
- Provides tips on how to evaluate a software component vendor

The purpose of this report is to provide guidance as you make these strategic decisions for your enterprise.

## Benefits of employing reusable components in software architectures

---

If you were constructing an office building, you wouldn't start by mining ore. Unfortunately, this is how traditional software design has been handled: each application starts with "materials" in their rawest form. Today, there is an easier, quicker, and more cost-effective way to develop software. Instead of coding each line manually, you can purchase software components. Like prefabricated walls in a building, components allow construction to begin at a higher level.

Components are prebuilt pieces of software that help engineers build an application. Some components are fine-grained, such as object-oriented class libraries and visual builders. Larger-grained components include database systems, application servers, and even prebuilt applications. Some components are used in the process of building an application, others are functional only at run time.

Businesses worldwide are gaining power and flexibility in their applications by developing them within object-oriented, component-based frameworks. This section details the many benefits your enterprise can obtain by using components in your software development projects.

## Reduced development time

It's a sad fact that most development projects are delivered late. But your team can break this pattern. Using software components in application design will drastically reduce the time to market for your applications. With components, engineers write fewer lines of code. Instead of "mining the ore" with each new application they produce, engineers can focus on architecture and business logic, slashing development cycles by weeks, months, or even years.

Here's an example: Gallo Wineries cut its development time by *two-thirds* when it used software components to develop a highly functional, asynchronous, messaging-based middleware application. The project, which previously would have taken one-and-a-half to two years to complete, was finished in just five months. Like Gallo, thousands of companies worldwide are discovering how software components can shorten development cycles.

## Increased developer productivity

In the object-oriented structures common to component architectures, data and the functions needed to manipulate and access data are grouped, or *encapsulated*, within self-contained units called *objects*. Objects carry data and a set of rules that tell other objects how to interact with them. The developer defines the relationship between these self-contained objects, but need not understand the internal workings of the object itself. Freed from mundane tasks such as establishing data structures, defining algorithms, and implementing database and network access, engineers can focus on the business logic of applications. This freedom from details helps engineers create more functional applications in shorter time frames. They'll have more time to work with key personnel on domain-specific issues, helping to ensure the success of customized applications.

## Application flexibility

In traditional procedural programming, functions that access and manipulate individual pieces of data are spread throughout the application. When changes to business rules must be made, engineers must locate each instance where functionality is affected, making maintenance a nightmare. The Y2K crisis is a perfect example of the weaknesses inherent in this programming methodology.

Your business must be able to adjust to the ever-changing computing landscape, which means you need a system that is flexible enough to adapt quickly as your business needs change. Object-oriented components provide this flexibility because code is not spread out, but isolated in a component. As a result, component-based applications are more adaptable to change, and your team will realize a dramatic reduction in maintenance overhead.

## Application reliability

Well-designed components manage their own memory, resources, and error handling, so engineers don't have to. This streamlining of the development process leaves fewer places for things to go wrong. Additionally, because each component has its unique job to do, the danger of duplicate functionality is greatly reduced, making revisions and upgrades simpler. And, since expert engineers have already debugged and tested the components, your team can concentrate its quality control efforts on the application logic. The result is cleaner, less error-prone code throughout the application.

## Component reusability

Software components are reusable, which means they give you a great return on your initial investment. Your one-time cost will diminish in relative terms with each new application that is built with reused components. A survey by QSM Associates, cited in the August 1996 issue of *Software Magazine*, found that organizations achieving high levels of reuse experienced a *900 percent increase* in productivity, a *70 percent reduction* in development cycles, and an *84 percent reduction* in costs. Although these productivity results were obtained over time as companies climbed the experience curve, they illustrate what can be achieved by including reusable software components in your application design.

## High return on investment

Reduced development time, increased developer productivity, application reliability and flexibility, and component reusability all play a part in giving you an incredibly high return on investment when your engineers use components in application design. See the section entitled "Calculating your return on investment" for illustrations of the cost savings that can be achieved.

## Risk mitigation

Component vendors know they must keep up with emerging standards in databases, compilers, and operating systems in order to remain competitive. Because of this, much of the maintenance your engineers are currently doing in-house can be handled by your component vendor. Once a solid, component-based framework is established, building new applications will take only a fraction of the time your teams previously spent. These factors help to minimize the risks inherent in new software design.

"The issue of predictability of software development can be scary," said Dr. Stan Malachowski, of Scientific and Industrial Software in Australia. "I've seen a lot of failures that are primarily due to poor estimating and the lack of risk mitigation. But I believe that both of these problems are ameliorated by adopting object-oriented technology and building on proven commercial components. I have consistently found that my ability to predict the outcome (time and money) of a software development venture is significantly enhanced by building over well-established frameworks."

## Calculating your return on investment (ROI)

---

To date, no definitive empirical study has been published on the ROI of reuse, so most of the data available is anecdotal, and based on individual projects or the experience of individual companies. But the available evidence points to significant savings for organizations using component-based development.

By using component-based development, engineers can create a common structure for at least 80 percent of the functionality for most enterprise applications. The remaining 20 percent of each application is the portion that must be developed by engineers intimately familiar with your business processes. As you can imagine, these percentages translate to some impressive savings figures in a development team's budget.

This section provides three different ways to examine component-based development in budgetary terms.

### Building a component in house vs. purchasing off the shelf

It is often argued that in-house components can be produced as efficiently as prepackaged ones. Engineers may be particularly reluctant to purchase a fine-grained component in an attempt to avoid the steep learning curves involved in working with someone else's code. Engineers also may express a preference for the "do-it-yourself" approach. Let's look at these objections individually:

- **Learning curves.** As with any new process, there is an initial learning curve involved in the transition to component-based development. But with each application built, the team's expertise in component-based architecture increases. In other words, the team begins to reuse its knowledge as it reuses the components. Also, engineers increase their own coding skills by working with prewritten code produced by industry gurus. Forward-thinking organizations recognize that this initial investment promises a long-term return.
- **Do-it-yourself.** This approach is not always cost- or time-effective. Junior engineers may need weeks or months to program and test basic low-level software components. Engineers with more experience, who could do the job more quickly, get bored writing low-level components. They prefer to work on more challenging pieces of the application. On the other hand, an investment in expertly written, commented, and tested components is quickly recouped.

Let's look at the in-house cost to create a single fine-grained component, such as a class library. An average developer, producing 100 lines of tested, usable code per day, can earn around \$100,000 a year in salary and benefits (about \$420/day). Let's work out the time and cost associated with developing a component that consists of 50,000 lines of code (a typical size):

Time to develop =  $50,000/100 = 500$  working days, or 2.1 years

Cost to develop =  $500 \times \$420 = \$210,000$

The time and costs increase when using less-experienced engineers. In this example, let's assume that the developer produces only 75 lines of code per day and makes \$75,000 in salary and benefits (about \$315/day):

Time to develop =  $50,000 / 75 = 667$  working days, or 2.8 years

Cost to develop =  $667 \times \$315 = \$210,105$

Given that class libraries sell in the \$500 to \$5,000 range, you can see how cost- and time-effective such an investment in components can be—even if only a few of the classes are used. And because these components have been tested and proven, you can rely on them with confidence.

## Lowering maintenance costs

In 1994, in the early days of the component-based development model, J. Bradford Kain reported in *Object Magazine*:

*“IS organizations can spend up to 80 percent of the total budget on maintenance. This is a significant drain on their ability to support new development or further automation of business processes. However, maintenance is a very broad term; it often means three different activities in a development organization: fixing errors in the current system, supporting new functionality required by the related business processes, and modifications due to changing requirements.... [E]ach dollar spent on new development will yield \$0.60 of maintenance each year during the lifecycle of the application. Approximately \$0.20 [is spent on] operational costs and the remaining \$0.40 is spent on maintenance activities.”*

The chart below illustrates Kain's projections of application development and maintenance costs:

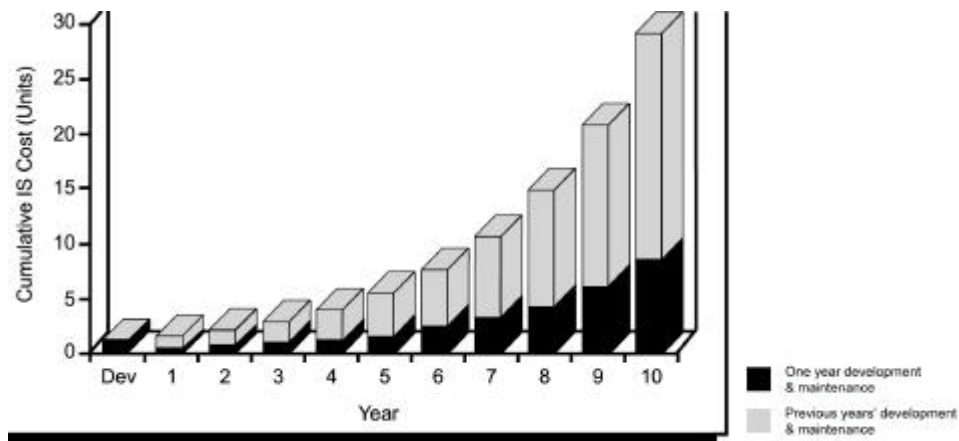


Figure 1. Cost of development and maintenance of an application.

Reprinted with permission from *Object Magazine*.

As mentioned previously, the use of object-oriented components will significantly reduce these expenses.

Let's look at Kain's three maintenance activities to see how component-based development mitigates these expenses:

- **Fixing errors in the current system.** Using components written by experts—and tested in the real world of enterprise computing—can drastically reduce errors in your applications. The more your infrastructure relies upon these types of components (remember that up to 80 percent is possible), the less error-prone your system will be.
- **Supporting new functionality required by the related business process.** Because of the modular nature of component-based architecture, new functionality is easier to add than ever before.
- **Modifications due to changing requirements.** Component vendors must support new operating systems, compilers, and development environments in order to stay in business. By allowing the components to handle the details of implementation, engineers are free to concentrate on business logic.

## Saving money by saving time

Imagine you're embarking on a development project estimated to occupy three engineers for a year. For this example, we'll assume one senior engineer at a cost of \$100,000 per year and two junior engineers at a cost of \$75,000 each. Your total personnel cost for development is:

$$\$100,000 + (75,000 \times 2 \text{ engineers}) = \$250,000$$

Now let's further assume that the utilization of reusable components allows you, like Gallo Wineries, to cut your development schedule by two-thirds. Your savings on personnel costs are:

$$\$250,000 \times .667 = \$166,750$$

...and your project is done in eight months instead of 24!

Keep in mind that this simplistic estimate covers personnel costs only, and doesn't include space and materials overhead. When you combine the reusability factor, the reduced maintenance costs due to increased reliability, and the increased ability of your organization to adapt quickly to change, you have a formula for significant return on your investment. Even if your experience is not as dramatic as Gallo's, this example serves to validate the magnitude of payback possible when you employ reusable components.

## Evaluating component features

---

Discussion of all component features would be impossible in this abbreviated format. Some of the features you look for will be highly specific to individual projects and to the configuration of an operating environment. A feature-by-feature evaluation is best done with input from the engineers who will work on a project. However, some major features warrant a short discussion here.

### Well-written code

The components upon which you build your enterprise should be well-crafted by masters in the art of component technology. Too much is at stake to establish your foundation on anything less than the best. A well-designed component combines a simple, easy-to-learn interface on the outside with encapsulated, powerful functionality on the inside.

Today, engineers are frequently hired for their business expertise in addition to their programming skills. Now, more than ever, these engineers need tools that are intuitive, and that hide the non-business complexities beneath the surface. This lets engineers concentrate on application logic instead of the complexities of the programming language or environment.

### Extensible code

*Source code* is code that has not yet been compiled to run on a particular system. The availability of this code allows engineers to compile as necessary to suit the needs of the project, and to derive from the prebuilt classes through inheritance to extend functionality. In other words, source code provides engineers with the best of both worlds: expertly written, richly functional code that can be used to extend or add power to applications.

### Scalable to the enterprise

If your enterprise hasn't outgrown—and been forced to replace—a business-critical application or software system, you've probably heard the horror stories from colleagues. Such a nightmare can cripple a business for years, if not crush it completely. Today's global marketplace opens almost infinite possibilities for your enterprise, but to take advantage of these opportunities and grow your business successfully, your software must be capable of unlimited expansion. This means scalability should be one of the first features sought in any component you consider purchasing. Components developed with object-oriented methods allow modular integration within software architectures, helping your applications scale as your enterprise grows.

## Portable or platform-specific

If you have standardized on Windows® for a particular development project, be sure to find a vendor that specifically designs for this platform, and preferably one that partners with Microsoft®. But if some of your development environment is heterogeneous, there is no need to waste valuable developer time porting the same component to different platforms. Instead, choose components with a portable object-oriented API. This will save months of coding time and effectively eliminate any concern over compiler inconsistencies.

## Support for internationalization

If your organization writes applications that handle multiple currencies, or that will be deployed internationally, you needn't code for each locale. Instead, use internationalized components, and gain:

- **Support for local formatting conventions.** While the U.S. standard is to format a date as mm/dd/yyyy, other countries standardize on dd/mm/yyyy, or yyyy/mm/dd.
- **Support for multibyte and wide character sets.** This is essential for Asian language support.
- **Support for the new euro currency.** Except for the Y2K issue, Europe's transition to a new currency has created the largest challenge IT departments are likely to face for some time. It is especially challenging while both the euro and local currencies must coexist. Components are available that can make this transition much easier on your IT team.

## Database-friendly

Undoubtedly, your organization relies on at least one relational database to drive key business processes. This means components required to interact with the database should be able to "speak its language." If your engineers are porting applications to your database from more than one operating system or platform, or if you have several brands of databases, porting issues can be enormous. However, using the correct set of components can virtually eliminate these issues.

Database-friendly, object-oriented components eliminate a procedural interface to relational databases. This allows engineers to port to a variety of platforms without rewriting a single line of code. Compiler and database inconsistencies become irrelevant with this type of component. You can also find components that solve compatibility problems between object-oriented coding methods and the relational model. And if your applications run in a TPM environment, you'll want components with substantial support for cross-platform recoverability and distributed transactions to protect data integrity.

## Interoperable between languages

Most engineers have discovered the value of object-oriented languages such as C++ and Java™. Yet nothing stays the same in computing, including the languages used to develop software. Although Java has been on the scene only for about three years, its impact on the industry has been significant. But another language may come along in a few years—or tomorrow—to augment or replace it. In addition, as information sharing becomes more crucial among heterogeneous systems, the contact points between languages increase.

This volatility in software design and usage means components must be interoperable. Engineers must be free to use the most appropriate language for adding the functionality that will drive your business forward. Object models that work with language translation layers make it possible to develop applications that involve clients, middleware, and application servers all written in different languages. A solution built on a solid, extensible, architecture will be ready to leverage new technologies, such as XML, as they emerge.

## Integrated within the object architecture

Establishing an object architecture that allows integrated design becomes increasingly important as your business moves toward rising levels of complexity, such as:

- Client/server and multi-tiered architectures
- Systems that need to run over the Internet
- Data exchange among a variety of platforms, operating systems, and languages

Because it's highly likely that your architecture will need to use more than one solution for optimum performance, this architecture must be able to support a variety of middleware, such as CORBA (Common Object Request Broker Architecture), COM (Component Object Model), and RPC (Remote Procedure Calls). These solutions handle the interoperation of objects throughout a distributed network.

For example, you might have Windows clients that need to speak to a Windows NT® server that uses COM, and to a UNIX server that uses CORBA. Without a component solution that can handle the difference in protocols, your development team must design patches to make things work. These “Band-Aid solutions” for interoperability are time-consuming, expensive, and likely to be inflexible, unreliable, and difficult to maintain.

If your engineers are using an IDE such as Visual C++®, Visual J++®, or others, be sure that the components you choose are integrated with the IDE.

## Thread-safe

As computing environments become more distributed, systems must allow parallel processing of computing tasks. Multithreading enables this parallel processing, and it is one of the most efficient ways to maximize the use of computing resources such as databases, files, peripheral devices, and network ports. If your development team plans to take advantage of SMP (symmetric multiprocessor) architectures—a technique that involves adding CPUs to increase computational power—your applications must be multithreaded in order to scale. Therefore, where it is appropriate to the component, multithread safety is an important factor to keep in mind.

## Evaluating a component vendor

---

Components form the backbone of your enterprise's computing environment, so you must be able to trust your vendor's ability to deliver well-designed, consistently coded, thoroughly tested, and proven components for your mission-critical applications. Failure of a single component can mean lost customers, so it's imperative that you choose your component vendor with care.

You've undoubtedly discovered the value of building long-term relationships with other businesses. In the component market, such relationships are almost indispensable. In fact, a wise investor in software components frequently will choose to deal exclusively with a single vendor, or a consistent group of partnering vendors. Let's look at some of the benefits of vendor standardization:

- Provides your development team with a consistent infrastructure, style, and development standard that will improve communication and reduce the need for extended white board discussions about protocol. This is especially useful for newly formed or inexperienced teams of engineers.
- Avoids the Band-Aid approach to application design by improving component integration.
- Allows the establishment of ongoing relationships with the vendor's technical support staff.
- Opens up opportunities for preferential pricing on a vendor's products and services.

This section describes the characteristics you should demand when choosing a software component vendor.

### Excellent reputation in the marketplace

Start by looking at a vendor's market share and its reputation in the marketplace. Does the vendor have a large customer base? Do your engineers know the vendor's name, trust the vendor's expertise, and, perhaps, already use the vendor's products? Have key people in the vendor's organization been in the development arena long enough to have gained a solid grasp of the issues? Can you find substantial evidence of the company's

stability and vision for the future? Affirmative answers to these questions will start you on the right track.

Look for a well-established vendor that has traveled the uphill learning curve of component-based development, and make sure the vendor enjoys solid partnerships with major industry players. Search the vendor's Web site for testimonials from satisfied customers. Talk to engineers who have used the vendor's products.

### **Architectural vision**

No one in the industry has a crystal ball for predicting long-term research and development needs. But those who recognize that the computing world is a changing world choose adaptable development solutions. Your vendor should be able to offer an extensible, open, non-intrusive architectural vision, Internet-savvy components, and a robust translation layer. By establishing a relationship with such a vendor, your enterprise will have room to grow and adapt to meet tomorrow's business challenges.

### **Breadth of solutions**

A good component vendor will provide a broad spectrum of software components that gives engineers everything they need to build applications within a heterogeneous computing environment. This includes tools for fundamental tasks, as well as solutions for interacting with a database, performing data analysis, building visual interfaces, interoperating between languages, and modeling applications. You simply can't afford to patch together your applications with components that can't communicate with one another. It is imperative to find a vendor that offers a broad range of integrated solutions.

### **Standards-aware**

Standards should make your life easier, not harder. Your component vendor must understand the issues surrounding standards compliance *without* limiting your choices. Components based on standards allow your team to be more productive. Standards-aware components also provide teams with a framework for working together efficiently, but allow the freedom to easily integrate new functionality as the need arises.

### **Non-intrusive implementation**

Beware of vendors who ask you to adapt to their proprietary solutions. Instead, look for one that gives you non-intrusive, policy-free solutions. If you're already heavily invested in a relational database or in other essential legacy systems, the solution you find should not compromise these investments, even if you plan to migrate to another system eventually. Also, make sure the vendor doesn't force your engineers to derive from a predefined base class. This limits flexibility in the choice of components, which can, in turn, limit available functionality. A non-intrusive implementation will allow you

to protect legacy investments, and allow your engineers to adapt applications as your business needs change.

## **Thoroughly tested products**

Two levels of testing should be considered when evaluating components. The first is the internal testing process the organization uses before the software is shipped. Ask the vendor about its internal testing methods. Ask about the qualifications of its quality engineering team. Make absolutely sure that internal testing is a part of the vendor's manufacturing process.

The second level of testing is the real world. You can't afford to stake the future of your business on a mere promise of performance, so be sure to ask the vendor for referrals. If a prospective vendor can't provide you with a solid list of customer referrals, don't waste your time.

## **Complete documentation**

In this world of constantly changing computer programs, you've probably come to appreciate the importance of clearly presented, well-indexed, and cross-referenced documentation. It's equally important for components to be well-documented. Good documentation includes lots of sample code that suggests coding strategies. Also look for the availability of online documentation. Online access to sample code saves time for engineers, who can instantly copy it from the Web. Keep in mind, however, that many engineers also like to use hard copies for reference while away from the computer, so it's best to have both online and hard copy documentation available.

## **Technical support**

Each software design team and development environment is unique. Your needs for technical support may vary widely from those of a vendor's other customers, or even between projects within your own enterprise. For this reason, you should take a close look at the support programs offered by a prospective vendor. Try to determine whether a vendor's technical support will meet your needs, both now and in the future.

There are several ways a technical support team can provide assistance to your engineers. Here are a few services you should demand:

- Availability of engineers, via phone or Web, to provide assistance with the initial installation
- Guaranteed response time
- Services such as writing code examples and providing expert help to debug your team's code
- An easily searchable knowledge base, available through the Web

- Web-downloadable software patches (so engineers don't have to wait to receive media through the mail)
- Reduced costs on major and minor upgrades

Be sure to ask whether the vendor offers a knowledge base or discussion list, as this is a sure sign that the vendor is interested in helping your team succeed. Resources such as these give engineers the opportunity to share information with other engineers who are using the vendor's products. This makes it an invaluable peer-mentoring tool.

## Consulting, mentoring, and training

The ideal vendor, in addition to providing you with reliable components and thorough technical support options, will offer a wide range of consulting, mentoring, and training services to help your team successfully implement components within your environment. This can include sending engineers onsite to provide help with the initial installation in complex environments, helping engineers to design applications, augmenting your staff for the short-term with a team of development experts, solving porting issues, and helping to train your staff in the most effective coding strategies.

Coding for today's distributed systems is a serious challenge for even the best minds. Don't let your vendor simply walk away after the sale. Make sure the product is backed with a solid support and services organization.

## Summary

---

Using reusable software components in the design and implementation of today's complex, distributed computing architectures is becoming critical to the success of your enterprise. This paper has provided you with ways to cost justify an investment in reusable software components, and has given you guidance in the evaluation of component vendors and their products and services.

The benefits of employing components include:

- Significantly reduced development time
- Increased developer productivity
- Flexibility and reliability of applications
- High return on investment due, in part, to the reusability of components
- Reduced costs on initial development and maintenance
- Risk mitigation

Important component features include extensibility, scalability, an intuitive interface, internationalization, ease of database connectivity, and thread safety, plus integration among protocols and components, and with your favorite IDE.

Because the future of your enterprise is at stake, use care when choosing a component vendor. The vendor should be well-known and respected in the marketplace, and be able to provide a breadth of solutions. Look for a vendor with architectural vision, and purchase components that are well-written, non-intrusive, thoroughly tested, commented, and supported with excellent documentation. Demand an accessible support staff that provides a wide range of services. Finally, seek to establish a long-term relationship with a vendor who will provide a consistent, reliable framework for your component-based development. Committing to a responsive vendor will help you to assure component integration, allow the development of ongoing relationships with the vendor's support staff, and provide leverage in price negotiations.

## References

---

- Anderson, M. "A New Way of Looking at Objects." *PC User*, 6 Sept. 1995: 104-106.
- Coffee, P. "Pick Targets Carefully Before Choosing a Tool." *PC Week*, 1 May 1995: 30.
- Daly, R. Number Six Software, Inc., Washington, D.C. E-mail correspondence with author, 18 Nov. 1998.
- Demeyer, S., Meijer, T.D., Nierstrasz, O., & Steyaert, P. "Design Guidelines for 'Tailorable' Frameworks." *Communications of the ACM*, Oct. 1997: 60-65
- Graham, I. "Moving to Object Architecture." *PC User*, 6 Sept. 1997: 93-96.
- Hendrick, S.D. "Components, Objects, and Development Environments." *International Data Corporation Bulletin*, April 1998: 1-4.
- Kain, J.B. "Measuring the ROI of Reuse." *Object Magazine*, June 1994: 49-54.
- Kara, D. "Build vs. Buy: Maximizing the Potential of Components." *Component Strategies*, July 1998: 22-35.
- Kozaczynski, W., and Booch, G. "Component-Based Software Engineering." *IEEE Software*, Oct. 1998: 34-46.
- Malachowski, S., Scientific and Industrial Software Pty., Ltd., Collingwood, Australia. Email correspondence with author, Nov.-Dec. 1998.
- Rogue Wave Software customer profiles, <http://www.roguewave.com/products/customers/gallo.html>. 1998.
- Ruber, P. "Object-Oriented World Speeds Up Application Development." *InfoWorld*, 3 Mar. 1997: 68.
- Sarna, D.E.Y. & Febish, G.J. "Developing with Lego Blocks: For Component-Oriented Development to Succeed, You Need a Staff with a Prepared Mind." *Datamation*, Dec. 1994: 29-30.
- Williamson, M. "Software Reuse," *CIO Magazine*, March 1997.

© Copyright Rogue Wave Software, Inc. 1999. All Rights Reserved.  
Rogue Wave and .h++ are registered trademarks of Rogue Wave Software, Inc.  
Java is a registered trademark of Sun Microsystems, Inc.  
Microsoft, Windows, Windows NT, Visual C++, and Visual J++  
are registered trademarks of Microsoft Corporation.  
All other trademarks are the property of their respective owners.

**Corporate Headquarters**

Toll-free: (800) 487-3217

Email: [sales@roguewave.com](mailto:sales@roguewave.com)

**Rogue Wave Software GmbH**

Telephone: +49-6103-59 34-0

[www.roguewave.de](http://www.roguewave.de)

**Rogue Wave Software B.V.**

Telephone: +31-20-301 26 26

[www.roguewave.nl](http://www.roguewave.nl)

**Rogue Wave Software S.A.R.L.**

Telephone: +33-1-4196 2626

[www.roguewave.fr](http://www.roguewave.fr)

**Rogue Wave Software-UK.Ltd.**

Telephone: +44-118-988 0224

[www.roguewave.co.uk](http://www.roguewave.co.uk)

**Rogue Wave Software S.R.L.**

Telephone: +39-02-3809 3288

[www.roguewave.com.it](http://www.roguewave.com.it)

