# Layering Frameworks in Component-Based Development

*Part of the Component Design Patterns Project*

Philip Eskelin
Ernst & Young LLP
750 Seventh Avenue
New York, NY 10019
+1 (212) 733-7638
philip.eskelin@acm.org

## Abstract

Many projects today use a component-based approach to developing software. Component-Based Development (CBD) stresses language and platform interoperability, and separation of interface from implementation. Existing and newly constructed components are being deployed to clients and servers to build flexible, reusable solutions.

Throughout the history of computing machinery, hardware and software engineers have endeavored to make it easier to utilize computers to solve problems. Many times, this has been done through raising the level of abstraction. And a consistent, exponential increase in the performance and storing ability of these computers has facilitated it.

Components and component frameworks are being used today to build solutions at a higher level of abstraction. However, successful integration and deployment can be a complicated task. This paper presents LAYERED COMPONENT FRAMEWORK as a pattern that helps framework builders build more cohesive and flexible component frameworks.

# Introduction

As with many other technology trends, industry analysts look into their oracles and promise that CBD is the big silver bullet. While it provides many benefits and can facilitate rapid delivery of successful solutions with a high return on investment, it's never a drop in the hat. Lack of solid project management, architecture, and design in CBD can be just as lethal as with any other technology panacea. Developing software in the context of CBD is not easy, but it can lead to highly successful results if done using proven techniques.

How do we make it easier for software developers to do it right? Let's take a step back and explore a historical trend we've seen in software development.

# Raising the Level of Abstraction

Throughout the history of computing machinery, hardware and software engineers have endeavored to make it easier to utilize computers to solve problems. And an exponential increase in performance with a consistent reduction in size and cost is what makes it possible. This has facilitated a trend of raising of the level of abstraction between the user and the computer to higher and higher levels.

Perhaps one of the first instances of this trend was punch cards. They made it easier for the programmer to quickly enter a batch of commands into a computer's operating system. Programmers went from directly controlling the machine's buttons and switches to feeding punch cards into a slot.

Over the last few decades much of this change has occurred in software development. Programming languages, compilers, software components, and frameworks continue to make it easier and easier for programmers to build applications and systems that are faster and simpler to build.

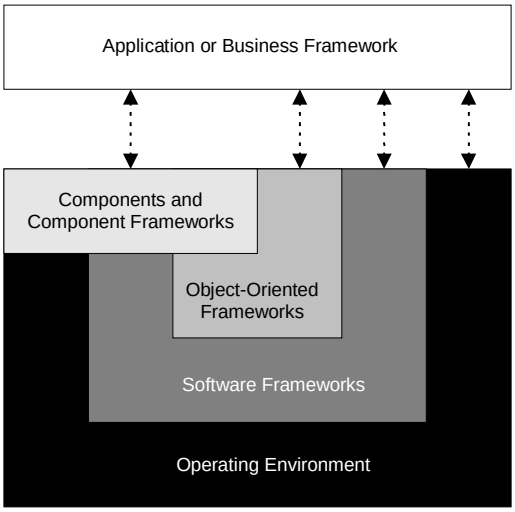
For example, in the 1970s, programmers commonly wrote code in assembly language. Then, more efficient compilers emerged that allowed programmers to write code in higher level languages such as C++ that were nearly as efficient as if they wrote it in assembly. Now programmers could craft solutions more rapidly that were easier to maintain and debug with an acceptable level of performance.

The trend was inevitable, driven by economic pressure and a consistently exponential increase in performance of computing and networking. In any software project, there is a tradeoff between the amount of programming effort, application functionality, and resulting performance. Better hardware means that there is more performance available to be spent on increased functionality and reduced development cost [Bolosky+98].

Over the last few years, it has been pushed further by CBD becoming a mainstream technique for reusing components and programming at a higher level of abstraction. Different programming languages and development environments are being used in CBD to assemble solutions from components and frameworks of all shapes and forms.

However a major drawback is that too many layers can make it more difficult than it is worth. It can increase the pain at assembly at a virtually exponential rate. How can one reap the benefits from a higher level of abstraction without entering an abyss of integration nightmares? The LAYERED COMPONENT FRAMEWORK pattern helps you answer this question.

# LAYERED COMPONENT FRAMEWORK*




… the patterns BUY WITH CARE and BUILD FOR THE USER can result in building or reusing frameworks that provide an abstract design for solutions to a family of related problems. Frameworks represent a large-scale form of reuse that can reduce development costs. Users of frameworks have requirements that often transcend its scope. This pattern describes a way of structuring a framework to simplify the architecture of an otherwise complex solution.

❖ ❖ ❖

**Frameworks should increase productivity and reduce development costs through large-scale reuse. Yet their builders often make assumptions that fail to meet user requirements. As a result, customization and integration become extremely complicated, making the situation worse rather than better.**

A single framework can be complex, expensive to develop, and difficult to use. Users might spend time finding clever ways to work around deficiencies in the framework because the framework doesn't support adequate customization or the ability to bypass it for direct

access to lower-level frameworks. Sometimes the components and interfaces they provides are functionally adequate, but other requirements are not met (e.g., performance), forcing the programmer to break out of programming by interface and program directly to a lower-level implementation.

Programmers might require special support for distribution, fault tolerance, data entry, market feeds, data storage, reporting, visualization, etc. Large systems typically require many of these features in their architectures.

Existing frameworks tend to support some or most of the problem domains that should be addressed, but rarely support all of them. Frameworks that attempt to provide all these features out of the box tend to make incorrect assumptions that make it extremely difficult to use.

For example, an order management framework might implement a simple security model based on users and groups that requires the user to enter a password at login which is stored as an encrypted string in a database. This might not be enough for an Internet-based order execution system that requires 128-bit encryption, authentication, authorization, and several different levels of access.

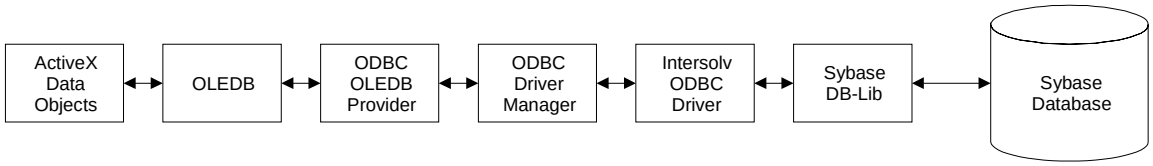
Other systems might perhaps not require security at all, or may need to integrate with an external enterprise security infrastructure. Framework developers cannot meet the needs of everyone without undue side effects.

A better approach is to build a component framework consisting of order management components layered on top of an application services framework that provides security, transactions, data access, resource and lifecycle management.

The application services framework abstracts the programmer from dealing directly with implementation details of core application services, and the component framework abstracts the programmer from dealing with common order management services. Together, they provide the same functionality. However, the layered approach reduces complexity and allows programmers to integrate custom components or bypass order management components altogether.

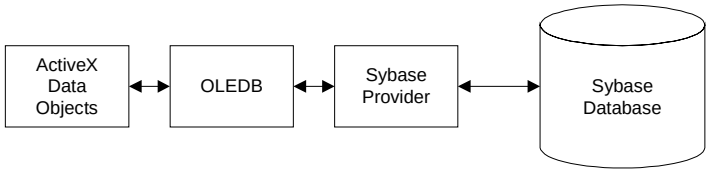
Rather than being faced with the problem of dealing with different requirements crossing many problem domains, framework builders are sometimes presented with the opposite: too many layers. They increase indirection, making integration difficult, and often perform insufficiently. Let us consider the Microsoft Universal Data Access architecture as an example. Programmers can use a component called ActiveX Data Objects (ADO) that provides uniform data access interfaces to relational

and non-relational data. In this example, the programmer is using ADO to access a Sybase database.




**Too many layers . . .**

ADO provides the highest level of abstraction in terms of uniform database access, but there are too many layers in between the programmer and the data. ADO goes through six different layers until it finally reaches the database. Many enterprises locate the database on a remote server. In addition to the above layers, several transport layers may be used to access it through a network. The integrator is forced to spend time ensuring that all six layers and the network are installed and configured correctly for the programmer to utilize ADO. A more direct approach is desired.




**A better approach**

A well-layered approach eliminates many of the excess layers by using a Sybase Provider that provides direct data access to the Sybase database. The assembler integrates less moving parts, and the programmer has the option of bypassing ADO to access lower-level components. Also, additional providers can be integrated that provide access to other data sources while allowing the programmer to reuse the same ADO interfaces for data access.

Therefore:

**Build higher-level frameworks in terms of lower-level frameworks. Each layer should provide a discernible and cohesive abstraction to lower-level frameworks. Give programmers who reuse the framework the freedom to bypass and customize it when necessary.**

❖ ❖ ❖

BYPASSABLE ABSTRACTIONS can be used by the programmer when

the component framework is unable to meet some requirements (e.g. functionality isn't suitable, unacceptable defects or performance issues exist, etc.). Framework layers will need to be integrated to act as a cohesive whole. Component frameworks define their own ABSTRACT INTERACTIONS. ADAPTERS [Gamma+95] can be used to interact with components in different frameworks that talk using different protocols. COMPONENT GLUE and MEDIATOR can be used to integrate frameworks or as an aid to assembling components together in a component framework.

# Conclusion

Future performance and storage capacity increases in hardware will continue to facilitate this trend of raising the level of abstraction even further. As software and component frameworks are built to futher abstract the programmer from the details of the system, allowing them to focus on the solution at hand, it is imperative that framework builders provide cohesive designs, assemblers integrate the appropriate layers, and programmers are allowed to bypass them.

# Acknowledgements

Comments and initial conception from Nat Pryce in discussions that led to the definition of the component framework came from discussions in the Component Design Patterns language project.

And thanks to Stuart Barker, Michael Feathers, Ralph Johnson, Scott Johnston, and Peter Maier for contributing to the Wiki Wiki Web discussion and evolution of the LAYERED COMPONENT FRAMEWORK pattern.

# References

[Alexander+77] C. Alexander, S. Ishikawa, M. Silverstein with M. Jacobson, I. Fksdahl-King, and Shlomo Angel. *A Patterns Language: Towns, Buildings, Construction*. Oxford University Press, New York, 1977.

[Bolosky+98] W. Bolosky, R. Draves, R. Fitzgerald, C. Fraser, M. Jones, T. Knoblock, and R. Rashid. *Operating System Directions for the Next Millenium*. http://research.microsoft.com/sn/Millennium/mgoals.html.

[Booch+99] G. Booch, I. Jacobson, J. Rumbaugh. *The Unified Modeling Language User Guide*, Addison-Wesley Longman, Reading, MA. 1997.

[Gamma+95] E. Gamma, R. Helm, R. Johson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley Longman, Reading, MA, 1995.

[Johnson+88] R. Johnson and B. Foote. *Designing Reusable Classes*. Journal of Object-Oriented Programming, Vol. 1, No. 2, June/July 1988.

[Johnston+98] S. Johnston and J. Gautier, *Layers of C++ Frameworks in ivtools*, http://www.vectaport.com/ivtools/ivtools-layers.html.

[Kara98] D. Kara. *Build vs. Buy—Maximizing the Potential of Components.* Component Strategies, Vol. 1, No. 1, July 1998.

[Szyperski97] C. Szyperski. *Component Software*, ACM Press/Addison-Wesley Longman, 1997.