JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2006**

Vol. 5, No. 6, July - August 2006

The API Field of Dreams – Too Much Stuff!
It's Time to Reduce and Simplify APIs!

Dave Thomas, Bedarra Research Labs

1 THE PROMISE

The benefits of well-designed class libraries and frameworks are a major selling feature for Object Technology. Further, the availability of class libraries in source code provides the benefits of open source, allowing developers to learn from the implementation, fix problems-, and make improvements through specialization via subclassing, delegation or, if absolutely necessary, direct modifications. Frameworks allow the definition of a domain-specific language. The packaging of classes with well-defined interfaces enables reuse in product lines and binary components. Standards such as OSGi promise components which can be assembled on the fly in devices ranging from phones to super computers. SOA once again speaks to the promise of thousands of services which can be discovered, composed and orchestrated.

Language and Tool Support

Modern languages such as Java and C# provide Interfaces which directly support the separation of implementation from usage as well as substitution of alternative conforming implementations, minimizing the impact on the client. Languages also provide control over the visibility and extensibility of base classes allowing designers to exercise at least some control over extensions of their classes. The ability to use meta data allows developers to embed pragmatics of the class design and intended use directly within the code.

Tool support enables continuous improvement. JIT compilation enables shortened compile times and the ability to quickly make a modification and execute a test case. Refactoring enables frameworks to be rapidly improved. Continuous Integration and Test is facilitated by JUnit and FIT, ANT reduces the risk of refactoring and provide the means of assuring code is both unit and acceptance tested as it is developed.

Cite this column as follows: Dave Thomas, "The API Field of Dreams – Too Much Stuff! *It's Time to Reduce and Simplify APIs!*", in *Journal of Object Technology*, vol. 5, no. 6, July-August 2006, pp. 23-27
http://www.jot.fm/issues/issue_2006_07/column3

2 THE REALITY

Unfortunately, the reality of frameworks and class libraries is far removed from the promise. With very few exceptions, OO libraries are bloated, poorly documented, very difficult to use and modify, and have disappointing performance. This accidental complexity makes using some of the popular frameworks challenging even for experts, let alone for normal application developers.

DLL hell has been replaced by API version hell! To use and extend frameworks requires intimate knowledge of the framework internals. The huge number of dependencies between class libraries results in bloated runtimes, and limits the ability to provide incremental software delivery. Once again this results in the need to ship the full boatload once a year.

Writing a simple application in J2EE or .NET requires an encyclopedic knowledge of hundreds of APIs. Just as developers become familiar with one set of libraries or frameworks, the rug gets pulled out from under them as a new, improved version of a competing framework appears. Should I use Struts, JFace, OpenAjax, RCP, or Swing/SWT? What about Spring vs. JBOSS vs. Websphere? Developers are paralyzed by choice, despite the fact that in every case there is substantial duplication between competing libraries.

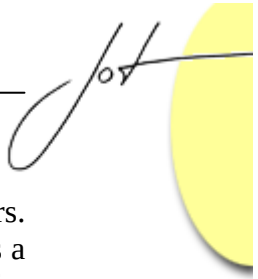
Despite the promise of SOA there are very few examples of true binary components which can be used as black boxes.

JSR Madness – The API Generator

While in principle the Sun JSR process is well intentioned and offers an opportunity for companies and individuals to work together, innovation and design is seldom a collective activity. Too often JSRs are a means to gain a control point as part of a business enablement or blocking strategy. JSRs with patents just below the surface make the use of libraries a potential license nightmare, e.g. Mobile JSRs.

Open Source – The Framework Field of Dreams

Each year a talented group identifies a problem in a complex framework and produces a (hopefully) better one. This creates endless framework wars as well as maintenance and evolution problems. Should we use Spring, JBOSS or pure J2EE? Should we use Swing or SWT? Should we use Hibernate or JDBC? We are trapped in framework hell! The problems become even more challenging when one assembles a new application consisting of different best-of-breed frameworks, which have different underlying component models, and may require different class library versions. Open source class libraries and frameworks present a configuration management challenge as old projects merge and diverge and new ones evolve.



It is clear that these problems are not uniquely confined to open source developers. One IBM product, which is built on the very respected and popular Eclipse platform, is a whale, containing over 2000 plug-ins, which will stress the RAM and clock speed of a high performance server, let alone a desktop.

3 WHY? CLASS LIBRARY, FRAMEWORK AND COMPONENT DESIGN ARE HARD!

It is clear from 20 years of experience that high quality class libraries, frameworks and components are hard to create, requiring very good people and lots of time. Typically 2 to 3 versions are needed before they are really ready for prime time. Rather than seeking minimalism and compactness, most libraries are over-engineered and full of unanticipated dependencies, thus forcing the inclusion of large wads of code.

Instead of a single, simple, consistent API, developers provide multiple APIs, helpers, and trivial and often useless extensions of base class libraries. Component Design is even harder because of the accidental and intentional coupling in frameworks. Despite the availability of language tools, far too often one finds OO code with few if any interfaces, just class definitions.

While much has been written about the design of frameworks, we have very little good work explaining how to use a framework properly. Frameworks have all sorts of API orderings, object instantiations, event sequencing, and thread assumptions which are seldom explicitly documented. Over time they are reverse engineered and documented in books and examples, but these quickly become out of date.

4 HOW CAN WE STRIVE TO REDUCE THE API MOUNTAIN?

It is surprising given the importance of API design how little has been written about the subject [1 - 5]. Techniques such as Design by Contract, Design Patterns, and Dependency Inversion provide best practices for designing and extending library frameworks.

KISS

Not surprisingly KISS applies to APIs as it does to most things. Take a “Just in Time” approach to API design, and try not to look too far ahead, anticipating future requirements that may never exist. Provide just enough function to solve the problem at hand and no more. Users will always let you know when you have left something out, but will seldom bother to tell you that you have included something they never use. It is important to provide only the API that is absolutely needed. Make each API prove to you that it is really required. Use Interfaces but don’t go overboard. Make a clear distinction about who owns/manages the objects, be it the client or implementer. It is important to provide one consistent way to do a thing rather than three different ways to do the same thing. This only creates more code to maintain and often confuses the user of your API.

Make it a goal to introduce as few new nouns and verbs as possible. The more names[1], the more the developer needs to learn and remember. Where a name is in common use but the meaning is different avoid overloading that name, use a different one which makes your intent clear.

Uniformity

Seek to leverage/extend existing API models which are familiar to the developer and which provide a more uniform API. There is a tendency to make each API very specific to a standard of some sort when in practice they are a special case of a more common generic abstraction. For example, use the lesson of Unix, Plan 9 and HTTP, and provide a simple uniform file system like abstractions for everything that can be made to look and feel like a file system. These APIs are already well understood. Why should there need to be very different WebDav, LDAP APIs etc. when a common directory/file API can be used and extended only where needed through additional parameters or special APIs. Relational and Collection APIs are also well understood and the same operations apply to many situations. As another common example, many APIs deal with selection, insertion or removal of objects from a tree or a graph. Why not use a generic tree/graph visitor API and a few element-specific operations? Developers already have far too much to remember, don't add to the burden with gratuitous invention. Generics and Iterators are language features assists the development of uniform interfaces.

5 SUMMARY

Don't fall victim to a closed, defensive programming style. Use an open and extensible design that allows downstream users to extend your code instead of having to invent yet another framework or API to do a similar thing. Achieving this goal will be more work, and undoubtedly require several refactorings, but it is well worth the effort. Build the services you need today, while trying to leave things open for future extension where needed.

Use the language supported documentation facilities to make it clear to extenders what how best to make extensions of your framework. While the code is essential it doesn't capture all of your assumptions. Provide a set of examples and test cases with your framework which clearly show its intended usage.

[1] Alan Kay is quoted as saying that creating new names should require a permit, recommending that every time a new class or method name is defined the developer should first be presented with a list of similar names and their dictionary definition.

REFERENCES

[1] Joshua Bloch, How to design a good API and why it Matters, OT 2004 Conference Keynote

[2] Bill Venners, API Design with Java, http://www.artima.com/apidesign/index.html

[3] Eamonn McManus, Java API Design Guidelines, http://www.artima.com/weblogs/viewpost.jsp?thread=142428

[4] Jim des Rivieres http://www.eclipse.org/eclipse/development/apis/API-First.pdf

[5] Java Collections FAQ, http://java.sun.com/j2se/1.5.0/docs/guide/collections/designfaq.html

[6] Krzysztof Cwalina, Brad Abrams, Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, Microsoft Net Development Series

About the author

Dave Thomas is cofounder/chairman of Bedarra Research Labs (www.bedarra.com), www.Online-Learning.com and the Open Augment Consortium (www.openaugment.org) and a founding director of the Agile Alliance (www.agilealliance.com). He is an adjunct research professor at Carleton University, Canada and the University of Queensland, Australia. Dave is the founder and past CEO of Object Technology International (www.oti.com) creator of the Eclipse IDE Platform, IBM VisualAge for Smalltalk, for Java, and MicroEdition for embedded systems. Contact him at dave@bedarra.com or www.davethomas.net.