Open Standards vs. Open Source

How to think about software, standards, and Service Oriented Architecture at the beginning of the 21st century

Robert S. Sutor, Ph.D.

rssutor@gmail.com

www.sutor.com/newsite/blog-open/

May, 2006

© Copyright 2006 by Robert S. Sutor. All Rights reserved.

Contents

© Copyright 2006 by Robert S. Sutor. All Rights reserved.

One: Standards

This year I blogged a wish list for the area of technology in which I work. The very first item was my desire for people to have a better understanding of the difference between open standards and open source. Authors have written entire books about these topics, and more are being written even as I pen this.

We saw in some of the public hearings in 2005 in Massachusetts about standardized document formats that there is some confusion between what a standard does and what the rules are when you go to implement it. I've been told secondhand that some people were even told things like "if you implement that standard, then all your software has to be given away for free." This, of course, was wrong and confused, if it wasn't, in fact, outright FUD (creation of Fear, Uncertainty, and Doubt).

So let's talk about some of the basic notions about and the differences between standards and open source. In this section, I'll focus on the former and start the discussion of software and open source in the next section.

A *standard* is like a blueprint. It provides guidance to someone when he or she actually builds something. Standards are not limited to software, but are an important part of computer hardware, telecommunications, health care, automobiles, aerospace, and many areas of manufacturing.

One of the reasons an architect produces a blueprint is that a builder, an engineer, or an inspector can look at it and say "if you build according to this plan, it will be safe and the house won't fall down." In the same way, some standards are for safety, especially where they involve electrical or electronic components.

A standard is more than just a blueprint, though, because it has to be something with which a lot of people agree. Something that may not have this sort of "blessing," or common agreement, is usually called a *specification.* By abuse of language, we will sometimes call a standard a specification. Put another way, all standards are specifications, but not all specifications are standards.

Standards are also employed when we have to ensure that things made by different people will either work together or work in the same way. I live in the United States, and when I go to the store and buy a telephone, I know that the telephone wire will plug into the jack in my wall. I don't need different jacks for phones made by different vendors. The design of the telephone jack and the plug are not control points for any phone vendor. I make my choice based on the features of the phone, the color, the price, whether it is wireless or not, and so on.

There are standards that describe the "blueprints" for the plugs and jacks, but the standards themselves are not the actual plugs or jacks. We separate the ideas of "a standard which may be implemented" and "something that is an implementation of a standard."

To bring this back to software, the OpenDocument Format is a standard, a blueprint, created by a technical committee at the OASIS consortium. It is not software, but rather a description of how you should write out the information in word processing documents, spreadsheets, and presentations should you ever need to put them on disk or, say, attach them to emails. The same description also tells you that if someone gives you a document, spreadsheet, or a presentation and they tell you it is in OpenDocument Format (perhaps via the file extension), then your software applications know what to expect when they read the file.

Because it is a standard, the information can be used by anyone who builds software that complies with the standard. No one vendor can arbitrarily change it, and that provides a lot of security for people who save their documents in OpenDocument Format.

Another important standard is HTML, the rules by which you format pages for the World Wide Web. Although there were vendor differences in HTML in the mid 90s, most people no longer think that it is reasonable to allow vendors to break interoperability by implementing too little of the standard or doing their own special things. We don't need different browsers

© Copyright 2006 by Robert S. Sutor. All Rights reserved.

to view web pages from different people, though some browsers like Firefox and Opera are known to adhere to the web standards, essentially the blueprints for the web, better than other browsers.

To extend my earlier analogy, we know that a web page ("the plug") will fit into the browser ("the jack") and then I can see and interact with the page ("I can talk on the phone").

Just as we said in the 1990s that no vendor should have a control point by having their own flavor of HTML, we are saying the same thing about document formats in the 2000s. In fact, since word processors have been with us a lot longer than the web, it's surprisingly that standards weren't created, if not demanded, there earlier.

So, to summarize, a standard is a blueprint or a set of plans that can be implemented. Where do standards come from?

A *de facto standard* is a specification that became popular because everyone just happened to use it, possibly because it was implemented in a product that had significant market acceptance. The details of this specification may or may not be available publicly without some sort of special legal arrangement.

The basic problem with a *de facto* standard is that it is controlled by a single vendor who can, and often does, change it whenever the vendor decides to do so. This frequently happens when a product goes from one major version to another. At that point, everyone else who is trying to interoperate with the information created in the owning vendor's product must scramble and try to make their own software work again.

This is easier, of course, if they can actually see the new specification and there are no impediments, legal or otherwise, to implementing it. The owning vendor gets a time-to-market advantage, possibly increasing its marketshare, again.

Traditionally, it was not in the interest of the owner of a *de facto* standard to make the details too widely available because they didn't want to make it easier for anyone else to move into their market space. They would say, "Why would I voluntarily let other people build products compatible with my data? They might steal away my customers!". To turn this around, it is not in the best interests of customers to be locked into *de facto* standards controlled by a single vendor. The customer might say, "I may have used your software, but it is my information, and I very much want and demand the freedom to use any application I want to process my information." *De facto* standards decrease customer empowerment and choice, though they linger on.

The second kind of standard I'll mention is something I'll call a *community standard.* As you might guess, this is something created and maintained by more than one person or company. The members of the community may work for companies or governments, belong to organizations, or may be experts who are otherwise unaffiliated. The standards creation process involves negotiation, compromises, and agreement based on what it best for the community and the potential users.

It is a classic fallacy to think that this necessarily creates a "lowest common denominator" or unsatisfactory compromise. Smart people can make good decisions together, even if they don't all work for the same company. Conversely, people who all work for the same company don't necessarily always make smart decisions. They might, for example, produce *de facto* standards that have security vulnerabilities.

Community standards usually get blessed, as I termed it above, by being created or submitted to a *Standards Development Organization,* or *SDO.* While it does happen that people may get together and write a standard from scratch in an SDO, it is very likely that one or more parties will bring drafts to the table as a starting point. It is usually expected that the developing standard will change over time as more minds are directed at the problem that the standard is expected to help solve.

A standard may go through multiple versions: it is not uncommon for the first version to take one to two years and then about the same time for each of the next one or two iterations. At some point it will stabilize and either become fairly universally used or else become eclipsed by an alternative way of tackling the same general problem. For example, the new web

© Copyright 2006 by Robert S. Sutor. All Rights reserved.

services standards are starting to be used for distributed computing, replacing older standards, as Service Oriented Architecture becomes more broadly deployed.

I want to return to this "community" idea for a moment. If you bring something to an SDO, you take a risk that others may change the specification, perhaps in ways that interfere with your product plans. One word: tough. Working within a community does not mean walking in and saying "I'm the king (or queen) and you can't change anything unless I say it is ok."

The value of creating a standard in a community is that products from different sources can work together to build solutions that solve real customer problems. If you can't compete by creating superior, higher performing, more scalable, more secure products and perhaps the services to give the customers what they need, then I would suggest you have problems beyond not controlling the creation of a standard.

What you are basically saying is "I can't win on a level playing field." Your customers might be interested in hearing that.

Some SDOs are *de jure* organizations: they have particular credentials in national or international settings. Some governments have laws that make it very difficult to use standards that do not come from *de jure* organizations. ANSI, ITU, and ISO are examples of *de jure* organizations while groups like the W3C, OASIS, and the OMG are usually just referred to as consortia.

Sometimes a standard produced by a consortium will be submitted and blessed by a *de jure* organization to make it more palatable for government procurements. Of course, *de jure* organizations, like all standards groups, must be very careful what they bless because they have reputations for quality and relevance that they hope to maintain.

The last topic on standards that I want to address is what it means for a standard to be open. I've spoken about this previously in the blog but, at the risk of repeating myself, let me say that I think we need to consider five aspects of standards and ask some important questions about each of them:

- How is that standard created?
- How is it maintained after Version 1.0?
- What is the cost of getting a copy of the standard?
- Are there restrictions on how I can implement the standard?
- Can I use just a part of the standard or extend it and still claim compliance?

In answering these, we need to think in terms of transparency, community, democracy, costs, freedoms and permissions, and restrictions.

- The more transparent the standards process is, the more open the standard is.
- The more the community can be involved and then actually is involved, the more open the standard is.
- The more democratic the standards process is, where the community can make significant changes even before Version 1.0, the more open the standard is.
- The lower the standards-related cost to software developers who want to use the standard, the more open it is.
- The lower the standards-related cost to the eventual consumer of software that happen to use the standard, the more open it is.
- When the licensing of the standard is more generous in the freedoms and permissions it provides, the more open the standard is.
- When the licensing of the standard is more onerous in the restrictions it imposes, the less open the standard is.

From these and perhaps other criteria, we should be able to come up with some sort of "Standards Openness Index." In the meanwhile, use them when deciding for yourself how open a particular standard is.

At this point I hope you have a good sense of what an open standard is. I also hope you understand there is a spectrum of "goodness" for several aspects of how a standard comes into being and how it might get used. "Open" has become a standard marketing term, so make sure you ask good questions of those who are trying to convince you that they are as or more open than the other guy.

In the next two sections, I'll talk about software development and we'll start to ask questions about openness in terms of software implementations. Just as we saw with standards, there is a spectrum of "openness" when it comes to the creation and use of software. Open source software is a large category, but I'll focus on some of the basic issues and how open source can be related to open standards.

One last thing to think about: near the very top of this discussion I talked about how a standard can be considered a blueprint. Do people ever freely give away things they make from blueprints? Does that make business sense? Did you ever sign up for a cellular phone service and get a free phone? Why didn't they charge you? Do they make their money elsewhere?

© Copyright 2006 by Robert S. Sutor. All Rights reserved.

Two: Software

In the last section, I focused on what a standard is and what it means for it to be open. I compared a standard to a blueprint and said that we separate the ideas of "a standard which may be implemented" and "something that is an implementation of a standard." Open source is a particular way of implementing and distributing software, enabled by legal language that gives a range of permissions for what people may do with it.

Just as we saw that there are a number of factors that can determine the relative openness of a standard, so too are there conditions that allow, encourage, or even demand behaviors from people who use or further develop open source software. I'm now going to look at traditional commercial software and then in the next section contrast that with open source.

I'm going to assume you have an intuitive sense of what software is, though we'll return to that in a few moments since it is important for understanding what one does with open source software. Software is what makes the physical hardware in a computer do its magic.

That is, software might be a web browser, an online customer sales tool, a video game, a database that supports your local bank, the reservation system for an airline, the invisible commands that respond to your actions in your handheld MP3 player, or even what controls how your automobile changes its mix of gasoline and air as you start to drive up a hill. These are just examples, and I'm sure you can think of many more.

You can also probably think of several different "titles" of software that are in the same category, such as *Halo, Doom,* and *Syberia* in the video game category, and *Microsoft Word, Corel WordPerfect,* and *Lotus WordPro* in the word processor category. These are all examples of commercial software: you pay money and you get a legal license to play the game or write a novel by using the software.

© Copyright 2006 by Robert S. Sutor. All Rights reserved.

The license gives you certain rights but also places certain restrictions on what you can do. The features and value of the software, the cost, and the rights you obtain determine if you pay for the license. Similarly, if you don't find the restrictions too onerous or unreasonable, then you might be inclined to agree to get and use the software.

Depending on how you look at it, some of the rights might be seen as restrictions. For example, if the license says "you have the right to use this software on one and only one computer," then you are restricted to not putting the software on two or more computers. From the perspective of the company that wrote and licensed you the software, this is probably very reasonable because the company (the "developer") wants to be remunerated for its efforts and its costs for creating the software.

The solution to running the software on multiple computers is easy: you make some sort of deal by paying more. Minimally, if you paid $50 for one copy then you might expect to pay $100 for two copies, $150 for three, and so on. You might be able to do better than that, particularly if you are dealing directly with the software provider and you need a lot of copies. Tell them you need 1000 copies and see what they offer you!

Ultimately, the provider of the software wants to get paid and even make a profit, and probably the more the better as far as he or she is concerned. It can be complicated to figure out how much to charge, but sometimes the market will decide for you.

For example, my son William has a handheld video game machine and games for that never cost more than $40. A vendor could try to charge more for a game he or she creates for the machine, but if potential customers think it is too much, then they won't buy it and the vendor won't recoup his or her development costs.

If the vendor charges too little and the game becomes very popular then he or she might be "leaving money on the table." That is, the vendor might have been able to generate greater revenue and profit by charging as much as the market would bear.

© Copyright 2006 by Robert S. Sutor. All Rights reserved.

Sometimes charging less for a video game can cause more people to buy it. If there is an aftermarket for things like game hint books, stuffed animals of the game characters, and a movie deal, then the software provider might significantly lower the game price to get as many people as possible playing the game. The real profit might come from the aftermarket products.

Might the price ever go to zero? In theory yes, but it rarely happens when the game is new. Nevertheless, it is an interesting idea to give a game away if you can generate a tremendous amount of money in other ways.

This sort of thing is not entirely unheard of: my family has bought boxes of cereal that had free DVDs in them and the local fast food restaurant has given away DVDs with certain meals for children. In this last example, if the DVD promotion is successful enough that it significantly drives up the sales of food, then the restaurant chain might end up paying quite a bit of money to the distributor of the film or films.

Business models around software therefore are more sophisticated then simply saying “I will get all my revenue from what I charge to license my software.” The software developer is interested in the total revenue, remember, even if not all of it or even the majority of it comes from licensing.

Thinking beyond the video game category, we can see other ways to make money in the software business. One is support. What happens when the user of the software has a problem? He or she might search the Web for an answer or, failing to find a solution, might call the software provider for assistance.

Is this assistance free? It depends. Sometimes there is free support for the first thirty or more days after you bought and registered the software. Another option is to give the user one or two free calls and then charge after that on a per phone call/time taken to resolve the issue basis. Yet another option is to charge a fixed fee for a year’s worth of support, though there may be limitation on how many times you can call.

© Copyright 2006 by Robert S. Sutor. All Rights reserved.

It’s not uncommon to have tiers of support levels, from something fairly minimal to something which is comprehensive. The more you get or think you are going to need, the more you pay.

There can be significant money in providing support. In fact, if the support just concerns how to use the software, there can be companies with no relationship to the original software provider that provide support packages. In this way, a particular piece of software starts to create what might be termed an “industry” that extends beyond the original developer.

What happens if there is fundamentally something wrong with the way the software operates, a “bug”? Someone with access to the original source code must go in and figure out what the problem is, find a solution, and then distribute either a new version of the software or something that fixes the specific area that is causing the problem.

If you do not have the source code from which the software was created you can: 1) wait until the original vendor decides to fix it, which may very well be the best solution for non-critical items, 2) find a work-around, that is, another way of doing what you wanted, or 3) switch to an entirely different application that does not have the problem. There can be many kinds of problems, but security and data corruption ones are especially serious.

If you had access to the source code for the software, could you fix it yourself? Maybe, or you might be able to find or pay someone to do it for you. Are you concerned that the provider of your software might not be in business forever and so you want the extra insurance of having the source code in case you need it eventually? Maybe.

You might sleep better at night, as they say, by having the source code but you also might sleep just as well if you had a vendor you trusted and knew was going to be around for a long time. You will need to decide for yourself. You should make this decision from practical business considerations, not political or ideological reasons. We’ll return to this in the next section.

© Copyright 2006 by Robert S. Sutor. All Rights reserved.

Another area where a software provider might be able to generate revenue is in services. If I decide to buy new accounting software, how do I get the information from my old system to the new one? How do I connect the new software to the customer relationship management (CRM) software that my sales team uses?

These tasks might be easy to accomplish but, then again, they might not be. You might have the expertise yourself or on your staff to do this work but, then again, you might not. To whom do you turn for help in making your systems work together? It's possible that the provider of your new software will do the upgrade and integration work for you, but they may not do it for free. There may be other service providers who can do the job better or at a better price.

Perhaps there was a company that put together the various hardware, software, and networking components on which you run your organization. It might be a good idea to pay them to do the transition to the new software. Again, there may be someone who provides value to you around the software who is not the original developer.

The more successful and widely used the software is in the market, the more service providers there will be. This can be a bit of a "chicken and egg" problem because the software might not become successful until there are enough service providers to help customers get the most value from it. This is one reason why software vendors like to have many business partners.

The partners may provide software rather than services. It's not uncommon for a partner to specialize software for use in a particular industry. For example, the partner may work in the legal industry and provide special templates that make it easier to produce a broad range of legal documents with consistent formatting. They might provide document management software so that the attorneys can find and retrieve documents.

In this case again, the original word processing software enabled others to develop an aftermarket. The aftermarket can further increase sales of the software, setting up a "virtuous

© Copyright 2006 by Robert S. Sutor. All Rights reserved.

cycle" in which everyone makes money and the software might achieve commanding market share.

I went through this to show you how the creation, distribution, and use of software can create economic value and revenue that extends beyond the initial price that is charged for a license. Anyone who only looks at the license cost is not thinking broadly enough about the big picture.

Software quality is also important. If a customer support application crashes often and loses important data, then it probably isn't too important that it is less expensive than a more stable product. Conversely, higher quality software that costs less than the competition deserves to gain significant market share. In particular, software with security and privacy problems can sometimes not be worth any price.

Software features can affect price, though it is not the total number of features that is important. Rather, are the features that people really need well implemented? Are they easy to use? If I only use 20% of the features of a piece of software and I can find another application that only implements the features I need and costs less money, should I go with that?

Before I leave this discussion about commercial software, I want to talk again about standards. Standards make things work together. In an example above, I talked about integrating a new accounting system with my sales' team CRM software. How exactly do you do that?

If both pieces of software come from the same vendor, it might have some special language and protocols that link the two together. The vendor will probably claim that this makes everything work more efficiently. This is dangerous.

Proprietary schemes for connecting software make it very difficult to substitute software made by one vendor with software made by another. This is exactly what the vendor that

© Copyright 2006 by Robert S. Sutor. All Rights reserved.

creates the accounting and the CRM software wants. This gives control of your system to the vendor and limits your choices. Again, this is what the vendor in question wants.

This situation is called “lock-in” and, as I said, is dangerous. You want the ability to choose the best software for your organization, no matter what the cost or who the provider is. These proprietary schemes can become the *de facto* standards I discussed in the first section.

When the industry agrees to common ways of having software from different providers interact, then you the user, the customer, the consumer, the citizen wins. To repeat myself for emphasis, open standards allow software made by different people to work together and gives you choice and control.

One of the ways in which software works together is by sharing information. It should be a hallmark of modern software that the way this information is represented should not be proprietary. You want the ability to use any software that you wish to interact with your own information.

The price of the software is irrelevant in this regard. More expensive software should not be accorded more leeway in using proprietary ways to represent your information. Similarly, software with greater market share should not force more restrictions on who and what can use your data.

Through this discussion I’ve highlighted several important aspects of software, and have focused on examples of software for which you pay for a license:

- The license fee can be larger or smaller.
- The license fee can affect market share.
- Market share might be increased by a lower license fee.
- Support is one way of generating revenue from software, and it is not limited to the original developers.

© Copyright 2006 by Robert S. Sutor. All Rights reserved.

- Service delivery is another way of making money in the software business and neither is it limited to the original developers.
- Open standards are important for allowing software made by different people to work together.
- The price of software is irrelevant when thinking about whether open standards should be used: they should. End of story.

What would happen if the price of the software was zero (in any currency)? What if you had access to the source code and could make any changes you wished? What if you could build new software on top of software created by others without paying them?

Is this anarchy? Will capitalism survive? How will we make any money?

We'll look at these issues next.

© Copyright 2006 by Robert S. Sutor. All Rights reserved.

Three: Open Source Software

I want to jump to the punch line even before I make some definitions: open source software is something that you need to consider very seriously.

This is true whether you are a user of software, a creator of software, a software distributor, a software integrator, a software solutions provider, or a venture capitalist. There is a difference, though, between seriously considering something and eventually doing something with it. Nevertheless, if you don't educate yourself about the possible role of open source in your business, organization, or government, you might be leaving yourself open to missed opportunities and cost savings, as well as increased competitive pressures.

You need to be able to accurately assess what is the right balance of proprietary and open source software that will optimize the results you are trying to achieve. In this section I'm going to look at open source software and some of the factors that you need to examine in your serious consideration of its use.

Fundamentally (and informally), "open source software" is software where you can see, re-use, and redistribute all or part of the source code, plus some fine print. There are two fundamental concepts here: what do I mean by "source code" and just what is this "fine print"? If you are a software developer, you know all about the former, and if you are an intellectual property attorney, the second is your specialty. For those of you who are one or the other or neither, I'll discuss both.

"Proprietary software" is usually made available in a form that will run on your computer, but you are not given the original material from which it was created. You cannot freely incorporate proprietary software in your own products, though you may be able to obtain some sort of fee-based license to let you do this. The basic idea here is that proprietary

© Copyright 2006 by Robert S. Sutor. All Rights reserved.

software contains intellectual property that was created by the software provider and that is not shared because it offers competitive advantage.

Licensing proprietary software to users for a fee is a long standing business model in the software industry. Licensing is not the only way revenue can be created, and it is often supplemented with subscription, maintenance, and support charges.

There are precise definitions of what is and is not open source software, and I think the Open Source Initiative should be your primary source of this information. I'm going to adopt a more informal approach to give you enough of an idea of what people talk about when they discuss open source. That said, when you develop your open source strategy and management plan, you need to be quite careful of the details, particularly the legal ones concerning licenses. Of course, if you are using proprietary, commercial software I hope you are looking at those licenses with a great deal of care as well.

There are people who think all software should be open source and there are others who think that no software should be made available that way. This has at times caused political, if not almost ideological, arguments about the nature of open source and its relationship to economic systems. I am not exactly neutral, but neither am I at either extreme. The ultimate choice of what to use belongs to the customer and consumer.

If open source drives innovation and increases competition, that's a good thing. If it allows businesses to be more creative in their business models as I described in the last section, then that is a positive effect. If proprietary software provides the functionality, scalability, security, and performance that you need, then it may very well be what you should use today. (This is with the caveat that proprietary software should not lock you into its use by avoiding the use of real open standards.)

The software industry is changing rapidly, so make sure you re-evaluate your decisions at least yearly, if not every six months. Open source projects may get better in that time, but so too may proprietary products, possibly spurred on by open source competition.

It is likely that your world will increasingly be a hybrid mix of both open source and proprietary software. You may not even know this is happening. For example, approximately 70% of all websites use the open source Apache web server. When you browse the web, even if you are using a proprietary operating system and browser, you are still employing open source to carry out your tasks.

“Code” is the set of instructions that makes computer hardware do things. Hardware has low level code built right into the circuitry. There are also special chips that can be altered dynamically to have different instructions: this type of code is called “firmware.”

“Software” is a general term that includes firmware, but most frequently means the very large class of sets of instructions that make general purpose chips made by IBM, Intel, AMD, Sun Microsystems, and others do a wide range of things. Using the same hardware, I can create and run software like video games, operating systems such as Linux or Windows, accounting software, word processors, web browsers, email programs like Lotus Notes, banking systems, travel reservation systems, databases, and many others.

Software can run on top of other software, such as a spreadsheet running on the OS X operating system. A macro in a spreadsheet is also an example of software, though it is relatively small compared to the spreadsheet software itself. Informally, we call software that runs on operating systems “applications.”

The term “program” is used generally and vaguely to refer to any software that accomplishes a particular task. People who create software go by many titles. Here are a few: programmer, software developer, software engineer, application developer, and coder.

Just as I am using a text editor to create and then save this document to my hard disk so that you and others can read it later, I can use a programmer’s editor to create the code for a particular piece of software and then save that on disk. This document is translated by your eyes (if you reading it) or your ears (if you are listening to it), and then your brain into something meaningful (I hope). In the same way, there are special programs that translate high level code into the low level form that some particular kind of hardware understands.

© Copyright 2006 by Robert S. Sutor. All Rights reserved.

My words in the text are the source for any direct understanding you may get from this document. The high level code is the source for the low level instructions that the hardware understands, hence the phrase “source code.”

Just as it would have been possible to write this article in French, German, Russian, Japanese or some other language and still have it understood by a speaker of those languages, there are different languages in which a software developer can create source code. In fact, the original source code for an application can be a collection of parts done in several languages.

To fix ideas, here are some of the names of some software programming languages: Assembler, FORTRAN, BASIC, COBOL, Lisp, APL, C, SmallTalk, C++, Java, Perl, C#, Python, JavaScript, and PHP. The languages tend to get newer as you move from the beginning of the list to the end. Some of these languages, for example Lisp and PHP, are very different, yet they nevertheless all are ultimately translated to instructions that are run on hardware.

Some languages are better than others for helping you accomplish certain tasks. PHP is better for developing websites than COBOL, but COBOL is better for developing traditional transactional enterprise applications like high performance banking systems.

I don’t think there will ever be a time when only one programming language is used by everyone for every kind of task. There are engineering reasons that determine why programming languages are created and used. There are also computer industry political and competitive motivations, such as the developer of a particular operating system who creates a language that primarily runs on that operating system and makes it difficult to move applications to other platforms.

As I mentioned above, an application may be created by combining several different parts. Why might I do this when I am developing software? One reason has to do with design: if I can factor a big application into smaller parts, or modules, and they have well defined ways in which they talk to each other, then it is easier to understand, create and fix the smaller sections, while having the whole fit together like a jigsaw puzzle.

© Copyright 2006 by Robert S. Sutor. All Rights reserved.

Modules can also be written by different people, sometimes living in different parts of the world or working in another part of an organization. For example, if I am an expert in writing code that draws and manipulates photographs on a computer screen, then it makes the most sense for me to focus on implementing modules that need that sort of function, rather than have me write code for something completely different, like databases.

Another good reason for using this modular kind of design is that I can re-use the pieces in different projects. The more I can re-use, the faster I can develop software. If I know that the individual modules were written and maintained by experts, I have greater confidence that the software I put together will be higher quality and that I am efficiently leveraging the work of others.

So software applications can be broken down into components called modules and they can also use collections of special routines that are stored in what are called libraries. For example, I might have a library that does fancy mathematical calculations. If the modules and libraries are available as open source, then many people can use them freely. They can also look at and possible improve the code and then share that.

This means that a community can develop around the code to create, maintain, and improve it. This benefits everyone who might be able to take advantage of the code in their own projects.

One important place to do this sort of thing is where standards need to be implemented. If I am creating a word processor, then it makes sense to have a module that knows how to read and write the OASIS OpenDocument Format (ODF), and it is the only part of the program that does this. The rest of the application can use whatever form makes sense internally to store the information from the document, but when it needs to saved to or read from disk, then that one module will be used.

If I am writing an extension for a web browser that needs to read and then display an ODF document, I can re-use that module if I have access to it and have permission to use it. If there is a problem in the module or the standard gets updated, I can change the module if I have

© Copyright 2006 by Robert S. Sutor. All Rights reserved.

access to the source code. If I have access to the source code, I can read it and become a better software developer by looking at how certain tasks are handled.

If I don't have access to the module or its source code, I have to write new code to process the ODF. With luck, I'll do it perfectly ... eventually, though it seems a waste of effort if someone has done it before.

If everyone uses the same software module to implement a standard and it is high quality software, then more people can develop applications faster that use the standard correctly. In this way, the easy availability of software can accelerate the adoption of a standard, with all the resultant goodness of systems that can speak to each other.

If support for a standard comes with an operating system or a software development environment, then many of the things I just described may be true. However, if the standards support is proprietary you probably will not have access to the source code and you may not be able to use that standard as easily on other platforms, such as different operating systems.

This trade-off may be acceptable, but you should accept it, if you do, after careful consideration. You will need to depend on the vendor for updates and fixes to the software at the pace he or she decides to provide it. You also need to be careful that the vendor supports the standard correctly and that there are not hidden security problems or special vendor extensions.

This can be hard to tell if you don't have the source code but, again, might be acceptable. You get to make the choice. If the vendor supports multiple operating systems and platforms, you are less likely to have the sorts of problems described.

If a vendor has a history of and reputation for delivering proprietary software that provides significant value and high quality, secure products with good support, this may outweigh the advantages, to you, of having the source code open. If, however, the way the vendor develops, licenses, and supports his or her software will lead you to being locked into his or

© Copyright 2006 by Robert S. Sutor. All Rights reserved.

her world with limited future choices and flexibility, then you should carefully consider the possible advantages of an open source solution.

In a somewhat round-about way, I’ve described software development and pointed out some situations where it would be advantageous to have access to the source code. There is more to it than just seeing the code or having it shared in some limited way by a vendor.

Open source fundamentally means that you can benefit from the work of others and, in turn, others get to benefit and extend your work. In this way, the software development community can make progress by working and innovating in a collaborative way. It’s the power of this community working in an open, transparent way that helps make this model of software development work.

Why would you not want to make the source code for your software open and available? Here are some possible reasons:

- Your software contains code from other people, and you don’t have permission to give their property away.
- You are getting strategic advantage, differentiation, and revenue from your software and you don’t want to give that up.
- Your customers are content with the current situation and business is good.
- You are operating in a niche market and there is no pressure to open source.
- There is no community of developers outside your organization who will help you further develop the code.
- There’s nothing you need from developers outside your organization and you don’t want to share what you’ve spent your own resources to develop.
- The less others know about your code, the easier it will be to keep them from moving to competitive products.

© Copyright 2006 by Robert S. Sutor. All Rights reserved.

- You’re embarrassed by the poor quality of your code (don’t laugh, it happens).
- You fundamentally do not believe that sharing source code is the right thing to do with intellectual property because it might cause damage to the existing business model of fee-based software licensing.

Here are some reasons why you might want to make the code available as open source:

- The code implements a standard that you want to be widely adopted.
- You’ve used other open source code and you want to give something back to the community.
- There is no special business advantage to keeping the code private and you want to make it available for others to use and learn from it.
- Your code contains some real innovations that you want to be widely used.
- You are very proud of your code and you want others to see and use your work, and then think you are a superb programmer.
- Your business model allows you to generate revenue in ways other than charging people who want to use your software.
- One of your competitors has a business model that does not allow him or her to generate revenue in ways other than charging people who want to use his or her software.
- The code was developed while you were working under a government grant, and you feel that you owe it to people to make it publicly available.
- By making your code available, it will allow others to innovate on top of the foundation you are providing. That is, you think that innovation will happen faster if others don’t have to “reinvent the wheel.”

© Copyright 2006 by Robert S. Sutor. All Rights reserved.

- You believe that all source code should be freely available in the same way that academic results have been shared for centuries.

Now, the fine print.

A particular open source project makes its code available under a specific legal license and this lays out the rights, restrictions, and responsibilities you have when you use the code. There are many licenses and the Open Source Initiative lists many of them. They are not all compatible with each other and you cannot and must not necessarily mix code covered under one license with code covered under another license.

The most widely used license is the GNU General Public License (GPL). While it is hard to quantify, it appears likely that approximately 70% of all open source projects use the GPL. Code that uses the GPL is referred to as "free software."

By its nature, the GPL makes new code that incorporates older GPLed code also use the GPL. That is, the GPL is somewhat self-propagating as code that uses it is picked up and re-used elsewhere. This is exactly as the authors intended.

The GNU/Linux operating systems use the GPL. You cannot charge others for a license to use GPLed software and you must make your source code available.

Another commonly used license is from the Apache Software Foundation. This is an open source license that does allow direct use of the source code within commercial products. Unlike the GPL, the Apache License allows "defensive termination": if you sue someone because you claim that the software infringes on one of your patents, then you lose the right to freely use the patents of others that are implemented in the software.

In other words, you stop having the right to use the software if you are trying to stop others from using it. Much of the open source software that implements the standards of the World Wide Web is covered under the Apache license.

© Copyright 2006 by Robert S. Sutor. All Rights reserved.

There are many other licenses but a handful of them cover well more than 90% of all free and open source software. I illustrated a few of the sorts of things that licenses can include but there are others. If you are planning to open source your code, study the licenses carefully. If you are planning to use open source code in a product, also study the licenses carefully and you should probably speak with an intellectual property attorney.

With care and some flexibility on your part, you can very likely accomplish what you wish to do. Treat license considerations very seriously.

In the first section I spoke about standards, in the second I discussed software and business models, and in this section I dove into what open source is in a way that I hope makes clear some of the differences from the traditional, proprietary development methods. I hope that I've justified to you the assertion from the very first sentence, that you must seriously consider open source software.

I want to emphasize that we are now and will continue to live in a hybrid open/proprietary world. I also fundamentally believe that whether we are talking about standards or software, we are advancing towards increasing openness.

We are seeking greater transparency into what our software does, how it works, and how it interoperates with software elsewhere in our own organization as well as those of customers, partners, and suppliers. We are looking to leverage the community to build a strong foundation of software and standards on top of which we can innovate.

Finally, we want greater certainty that vendors will not be building things into software that benefit themselves at our expense, particularly if it limits our future choices of what applications we use to access the data that we ourselves created.

In the next section I'll tie all these threads together and talk about how and why Service Oriented Architecture, or SOA, is accelerating this movement towards openness.

© Copyright 2006 by Robert S. Sutor. All Rights reserved.

Four: The SOA Connection

In the beginning, there was one computer and it was big and slow and it filled an entire room. Eventually, there were many computers and they were smaller and they could talk to each other.

All was not good, however, because they did not speak to each other in the same way. Strangely enough, this was often true even when a particular collection of machines all tried to work together in a hospital, an automobile manufacturer, an insurance company, or a government agency.

In the meanwhile, the software that ran on the computers got bigger, more powerful, and sometimes needlessly complicated. If two computers did communicate with each other to get some job done, it was very difficult to put in similar software created by someone else on one of the machines. This was true even if that new software might have been significantly better in some way or less expensive.

It became very hard to substitute in different hardware that was much faster or otherwise better suited to more efficiently accomplish the intended task. We also discovered that machines started to know too much about each other. This was not nosiness on their part, of course, it was just that people started to depend on particular special features of the software or the hardware when putting everything together so the machines could do their jobs.

There were times when we wanted to use knowledge of special features in very high performance situations, but for most situations with application software, building in these kinds of dependencies eventually caused more problems than they were worth. What we really needed was a way for the computers, really the software running on these networked machines, to be able to ask each other for information or to do certain jobs in ways that did not give away the underlying details of the software, the operating systems, or the hardware.

© Copyright 2006 by Robert S. Sutor. All Rights reserved.

If you could do this, and computer scientists refer to such systems as being *loosely coupled*, then it would be much easier to completely hide the underlying details of how the systems were built. This would allow us to make changes or improvements to the systems while still allowing the software to communicate in the same way.

We could make the overall job run faster by putting in speedier hardware running a different operating system, yet the systems could still communicate in the same way. We could move one computer closer or farther away, speed up the communication technology, and yet everything would still keep working.

If a car manufacturer needed to order parts, then it could use exactly the same language and communication style to talk to two or more suppliers. If a new supplier offered better quality or a lower price, then that supplier could be added into the system and the same kinds of interchanges could take place with it as had been happening with the older suppliers.

What we're describing here is *interoperability*, software and hardware systems made by different people that can nevertheless communicate in a high level way that does not depend on the underlying implementation details. This means we don't all have to buy our computer hardware from the same vendor and we don't all have to use the same operating system and applications.

It means that we have the *choice* to buy or build or otherwise obtain what is right for *us* and it gives us *control*. This means that we don't have to rely on proprietary software communication schemes from vendors and we don't have to get software interoperability via one vendor's trade secrets.

A vendor or a software provider gets our business if they offer the best product, code, or service at the right price. They know we can substitute in something made by someone else. Hence we get more competition and ongoing improvements, both technical and economic.

The world doesn't quite work this perfectly now, but it could come close. We can build open, interoperable software systems useful for businesses, governments, schools, hospitals and

© Copyright 2006 by Robert S. Sutor. All Rights reserved.

anything else that could benefit from the advantages discussed above. Standards can make it all work together while open source and traditional proprietary software will give us a range of choices in how we build the systems.

Let's stop and think about the World Wide Web. When you use the Web, do you ever worry about what software is being used to deliver the pages you view? Do you think about the hardware?

Websites often run both proprietary and open source software and they use hardware from many different vendors. From the perspective of a consumer of web pages and someone who sends personal information on the Web, you want the transmissions to be fast, reliable and secure. You want to be able to fully interact with the pages within your browser, no matter which browser you choose.

That is, you care about the *quality of service* and you care about the standards being used to encode the pages and the way they are sent back and forth. This is possible today and has been for many years.

Recent releases from open source browser software providers and smaller proprietary browser software vendors has focused the spotlight on the importance of good, current support for standards and consistent attention to security issues. As a result, the market leader has been forced to update its browser in order to try to remain competitive and stop losing market share. Standards are like that: they force vendors to support interoperability in the way customers demand.

The Web illustrates the success of standards in hiding how sites are actually implemented. This allows a website owner to use any software or hardware that suits his or her purpose. That owner also wants good quality of service. He or she wants happy customers who are pleased with using whatever services are offered on the site.

Users of the sites don't have to think about the technology being used but rather can think about whatever they are trying to accomplish. This may be buying music, finding the local

© Copyright 2006 by Robert S. Sutor. All Rights reserved.

movie listings, ordering presents for an upcoming holiday, or reading blogs. There are many such examples, but in all of them the standards that make up the web allow you to think about *what* you are trying to do versus *technically how* you are doing it.

It also enables geographic independence. To give you an example, I have a personal website and I don’t have the vaguest idea where the machine running it is physically located. As long as interactions with it are fast enough, the location makes no difference.

This allows a lot of flexibility for website owners, especially the ones who may be running hundreds if not thousands of hardware servers. As long as the quality of service—performance, reliability, and security—is sufficiently high, it makes no difference where on the planet the hardware is. The Web is loosely coupled and its success is evident to hundreds of millions of people every day.

If we can accomplish all this with the Web and what it has done for e-commerce and making information available globally, can we do something similar but a bit more sophisticated and more general for interactions between arbitrary pieces of software? Can we have more fine-grained security where we can allow doctors to digitally sign the different parts of medical records for which they are responsible? Can we encrypt different parts of purchase orders so that only authorized people can see information relevant to them in a business process?

Can we easily substitute in new supply chain partners without disrupting our ongoing business and workflow? Can we transparently link multiple hospitals together so that all the electronic services we need to treat patients are available? Can a government provide the necessary infrastructure to take care of its citizens’ needs while being able to use open source or proprietary software ... or both?

Many people, including myself, think the answer is yes, and the way to do is via something called *Service Oriented Architecture,* or *SOA.* Open standards are what make it work.

© Copyright 2006 by Robert S. Sutor. All Rights reserved.

A *service* is something that does a particular set of activities and has a consistent interface. Think about an ATM, an automatic teller machine. I have used these all over the world and but for language and currency, they all pretty much do the same thing.

You put in your card, type in your personal identification numbers, and then you can interact with your accounts. You can transfer money from one account to another, withdraw money in the local currency, and ask about how much money you have. There are also other activities, but we would summarize all them as being "banking services."

Because of years of experience, there is a good and common understanding of what the standard banking services are. In fact, if you use the Web to do online banking, you will also be using some of the services. (I don't know of any that allow you to withdraw or print money from your computer!)

Whether you are using an ATM or using online banking, the steps are similar. You authenticate yourself (that is, you provide enough evidence that you are who you say you are) and then you invoke one or more services. You may use the balance inquiry service before you use the money transfer service. Eventually you do everything you want to do and you end your session. The next person who uses the ATM or your computer does not continue using your identity to access your accounts.

When you use an ATM, do you have any idea how the back-end banking systems are built? Do you care?

I maintain that you care about how quickly and robustly the ATM responds to you and that your privacy is maintained. This is quality of service again. You also care about successfully using any ATM you may encounter. This is standardization reappearing.

We can now translate these two banking ideas—ATMs and online banking—to computer to computer interactions. Using services, we could build software that automatically paid bills once a month without any human interaction once it was set up. We could automatically transfer money from a parent's checking account to her college student child's savings account in a different bank on the other side of the country. We could do all this without knowing the exact details of the service implementations.

To be clear, we would have to have the right authorizations to do any of this, but I'm talking about technical feasibility. For the sake of brevity of discussion, I'll assume from now on that the appropriate security is being used with any service.

If we think about health care, we can come up with other services that might be useful: ordering drugs from one or more pharmacies, requesting lab tests or their results, retrieving medical records including subsets such a lists of allergies, and so on. For government, we might recall traffic violation or arrest records, request local tax records to compare with federal ones, or provide real time epidemic status information.

For the travel industry, there might be services that query hotel availability, make airline reservations, book restaurants, and reserve theater tickets. These individual services could all be combined into a compound service that might be called "book my next vacation." I encourage you to imagine other software services that might be useful in various industries, your business, and your life.

The "orientation" part of SOA means that you try to use services for everything possible and practical. The "architecture" part of SOA means that you have some discipline and governance in how you design, create, run and maintain the services and how they all fit together. Services will increasingly be the way we implement the components of our business processes. They help us once again separate the "what" is being done from the "how."

The standards that we use to make this work fall into three categories: data formats, protocols, and interfaces. When we talk about software interoperability, this is what we mean.

A data format is how we represent the information we send back and forth. A protocol wraps up that data with the necessary transmission and security information so it can be moved reliably from one computer to another. The interface is the exact specification of how you tell

© Copyright 2006 by Robert S. Sutor. All Rights reserved.

a service to do something, whether it is a query or an action to be performed. All together, these three things describe how you talk to a service and how they talk to each other.

Data formats can be highly structured information such as the details of a banking transaction expressed in XML or something less structured like a doctor's notes contained in an OpenDocument Format memo. Protocols and interfaces are typically very structured. The standards being developed in the W3C and OASIS for web services are an important way of implementing SOA for many people.

If one vendor, individual, or group owns any one of these, there is a potential problem. We want the freedom to call any service we wish if we have the right authorization to invoke it. If a vendor prevents others from using a particular format unless they pay a fee, then there is effectively a tax on the communication between software services.

This was a major fear when the Web was maturing but luckily it came to naught, though it was not without some significant challenges. If a vendor requires software interfaces to be licensed then they are effectively trying to lock users into their way of doing things. When protocols are proprietary then we limit a customer's ability to link together software systems and services in ways that they choose.

In short, we need truly open standards and not vendor controlled or dictated specifications in order for SOA to reach its full potential as a solution for customers.

If I am insisting on open standards for SOA, is there any room for doing anything proprietary here? Yes, and that is in how the services themselves might be built. Since we are using open standards to communicate to and from a service, we have the freedom to implement the service using any hardware or software that we choose. The implementation will not affect what the user sees or does other than how it delivers its quality of service.

If we want to use open source software and it gets the job done in terms of features, cost, and maintenance, great! If proprietary software gives you the security, performance, scalability

© Copyright 2006 by Robert S. Sutor. All Rights reserved.

and ability to run on multiple hardware platforms, use it! If you use a combination of both, that’s just fine as well. That’s your choice and it is under your control.

Service Oriented Architecture is now a major driving force in the IT world. As we redesign our older software to operate in this new SOA environment, the value of truly open standards is becoming more and more clear.

Collectively we’re getting a better understanding of how open standards provide the freedom we need to factor our systems in the right way. This will allow us to openly interoperate with the software systems of our customers, partners, suppliers and, in the case of governments, citizens.

Open source is playing a role here because it is often how standards are first made available. SOA presents new business opportunities and better ways for industries to communicate within and between themselves.

There’s a healthy future for software development, be it open source or proprietary, and I believe open standards will be at the core of our success in the days to come.

© Copyright 2006 by Robert S. Sutor. All Rights reserved.