How to Use More Open Source

In Your Next Federal IT Acquisition

-- Robert L. Read, PhD, and Eric Mill

The history of open source software is a record of steadily turning tremendously expensive custom-built solutions into freely available infrastructure you can simply take for granted. What once were astoundingly sophisticated, expensive human endeavors have become open source tools you can drop into place in your project on a whim.

Open source has changed what it means to make software today. It's important that everyone seeking to build or procure new IT projects understand that open source exists, that it can be of high quality and highly reusable, and how to use it securely.

In this post, we'll lay out the story of how software tools become open source commodities, how to make use of open source in your projects, and how to think about the security dynamics of open source software.

Understanding Commoditization Trends

The Federal knowledge worker who shares responsibility for an IT project cannot escape the need to understand trends in the commoditization of information technology.

By commoditization, we mean the inexorable evolution of technology: from a unique idea, to custom-built usage, to a broadly used product, into an open-source commodity (which may be commercially supported). By commodity, we mean a product, function, or service broadly and cheaply available.

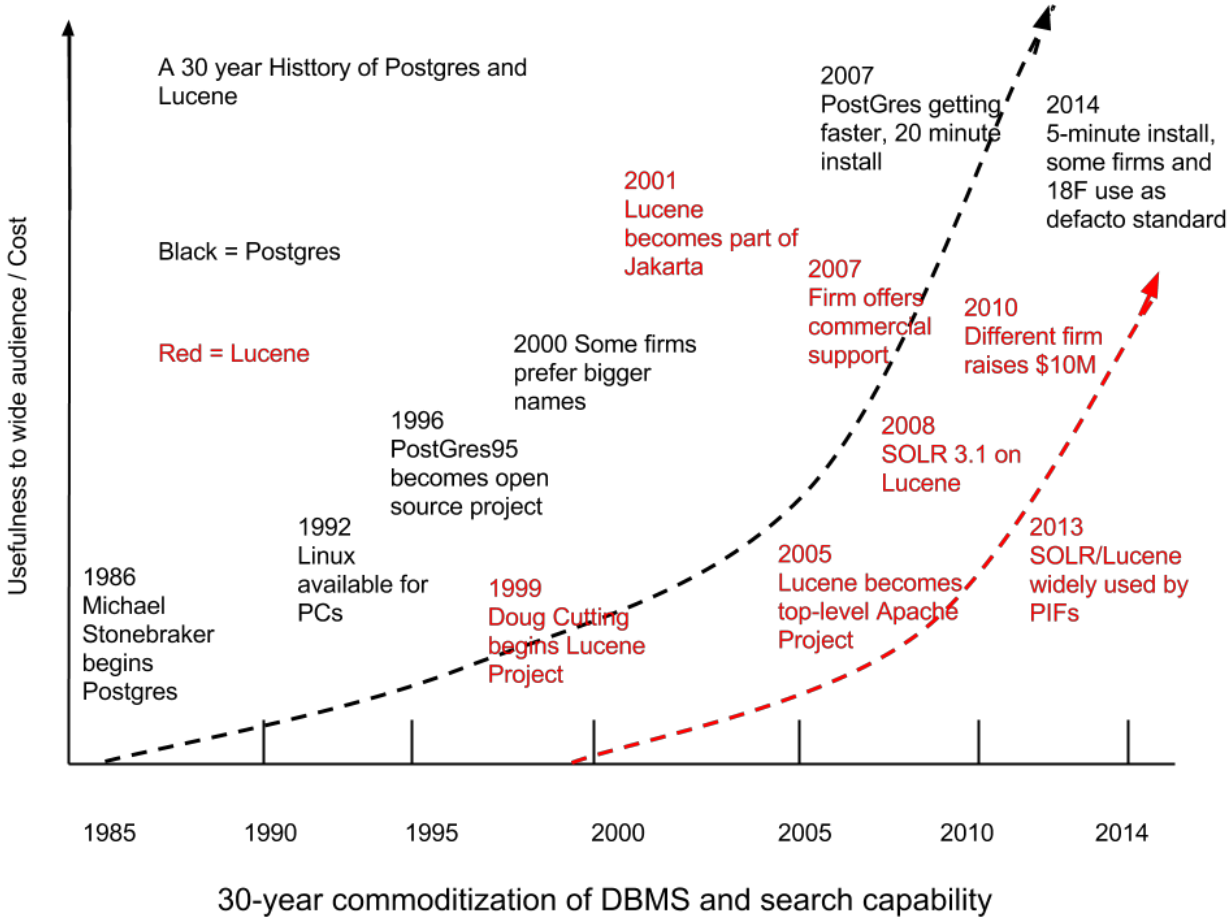
For valuable computer programs, the end point of this evolution is always freely available, reliable, well-supported open source software. Let us examine this process by considering one of the most important tools in information technology, the relational database system.

The Commoditization of Relational Databases and Full-Text Search Engines by Example

We can mark important events in the evolution of the relational database for end-users:

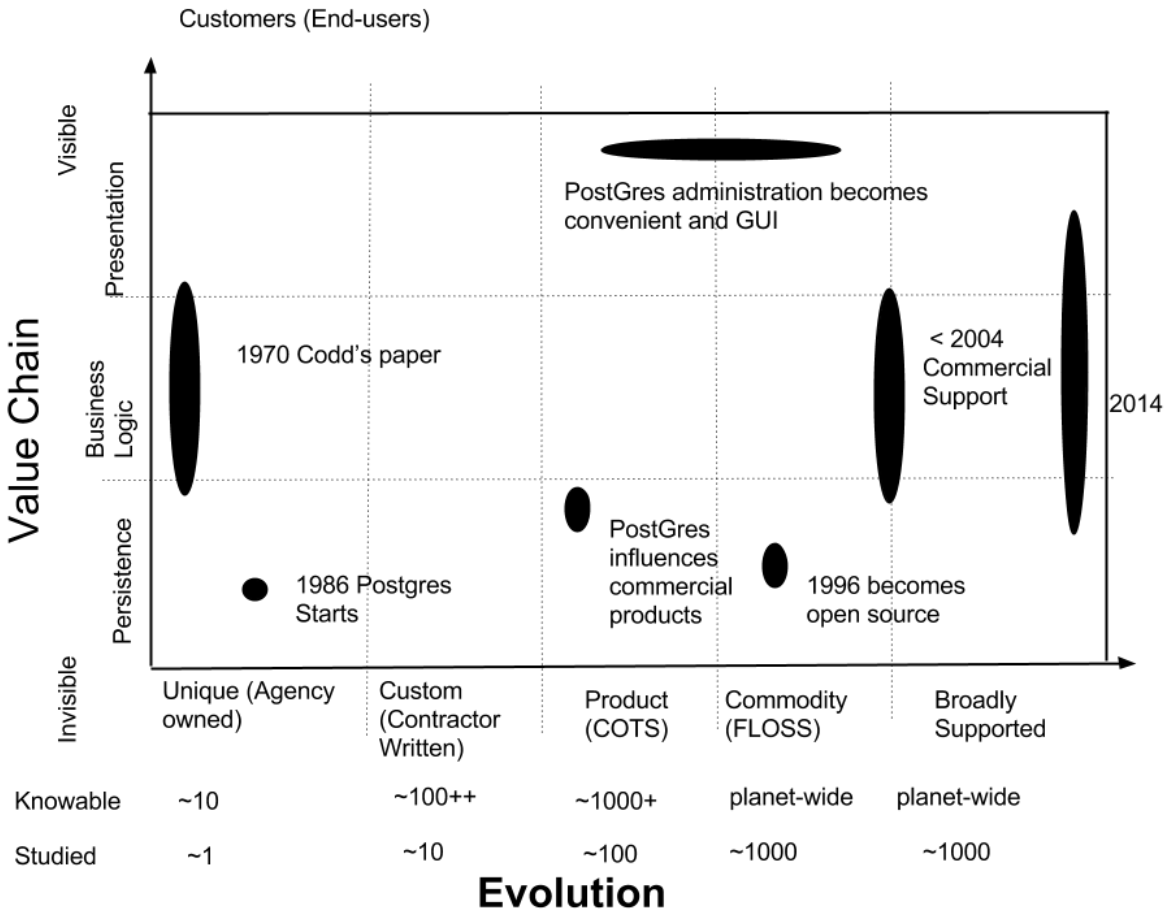
1. 1970 Codd published "A relational model of data for large shared data banks".
2. 1985 Michael Stonebraker begins Postgres project.
3. 1992 Linux becomes available on x86 architecture.
4. 1996 Postgres95 becomes an open source project.

5. 2004 At least one firm provides commercial support.
6. 2007 PostgreSQL either comes with a Linux distribution or can be installed in 20 minutes.
7. 2014 PostgreSQL can be installed in 5 minutes, is more convenient to use, and is the common backbone of 18F's projects and many commercial firms.



30-year commoditization of DBMS and search capability

Figure 1 shows our subjective evaluation of the usefulness of PostgreSQL relative to its total cost of ownership. This "geometrically" increasing usefulness is familiar from hardware performance curves plotted against time, but in this case owes very little to that phenomenon. Rather, it represents the typical commoditization of software. The community of users and developers drive important software tools from unique inventions to broadly available tools. Along the way, the tool usually becomes a commercial product, and then becomes available as an open source alternative.



The same diagram also plots the even more rapid evolution of a an open source full-text search engine, called Lucene.

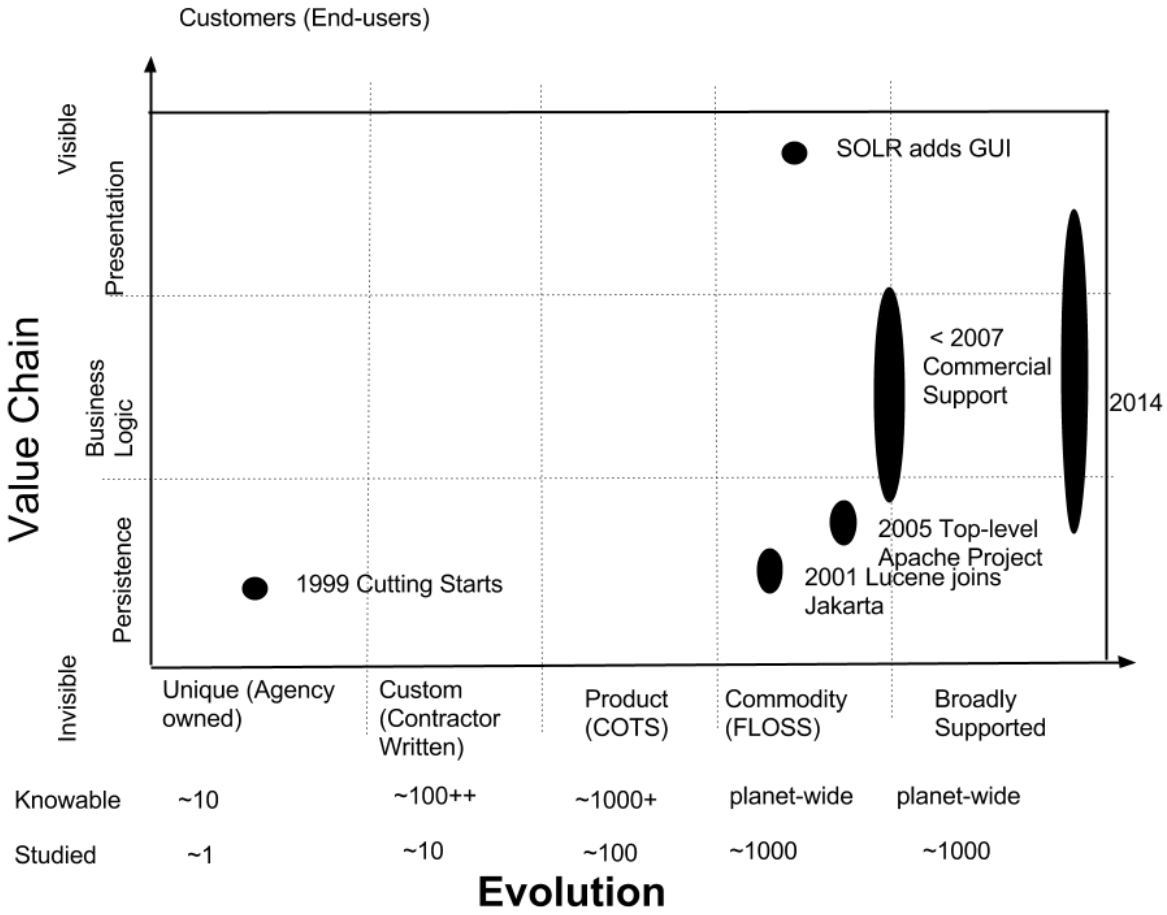
1. 1999 Doug Cutting writes Lucene and makes it available at SourceForge.
2. 2001 Lucene becomes part of Jakarta.
3. 2005 Lucene becomes top-level Apache Foundation project
4. 2008 SOLR 1.3 released
5. 2007 A firm begins providing commercial support for Lucene, SOLR, and related technology.
6. 2010 A different firm raises $10M in venture capital.
7. 2013 SOLR, ElasticSearch widely used by Presidential Innovation Fellows.

Simon Wardley has articulated and popularized the Wardley-Duncan map, a visual representation of evolution and value for general systems. Because we are concerned primarily with software systems and open source systems in particular, we specialize the Wardley-Duncan map to have named subdivisions of the Evolution axis which reflect the general evolution and commoditization of software.

We imply that software naturally evolves through stages, which in a Federal context we name:

1. Unique/Agency Owned,
2. Custom/Contractor written and closed source,
3. Productized into Commercial Off-the-Shelf software (COTS),
4. Commoditized Free-Libre Open Source Software (FLOSS),
5. Supported Commoditized, meaning that it either has commercial support or that it has a very large support community making its use practically risk-free and very convenient.

Furthermore, we impose on Wardley's Value Chain axis the [standard 3-tier architecture], consisting of a Persistence Layer, Business Logic Layer, and Presentation Layer.



We can now use this modified chart to understand the evolution of PostgreSQL (Figure 2) and Lucene (Figure 3) in greater detail, and therefore the evolution of Relational Databases and Full-text Search Engines.

How to use the Wardley-Duncan Map

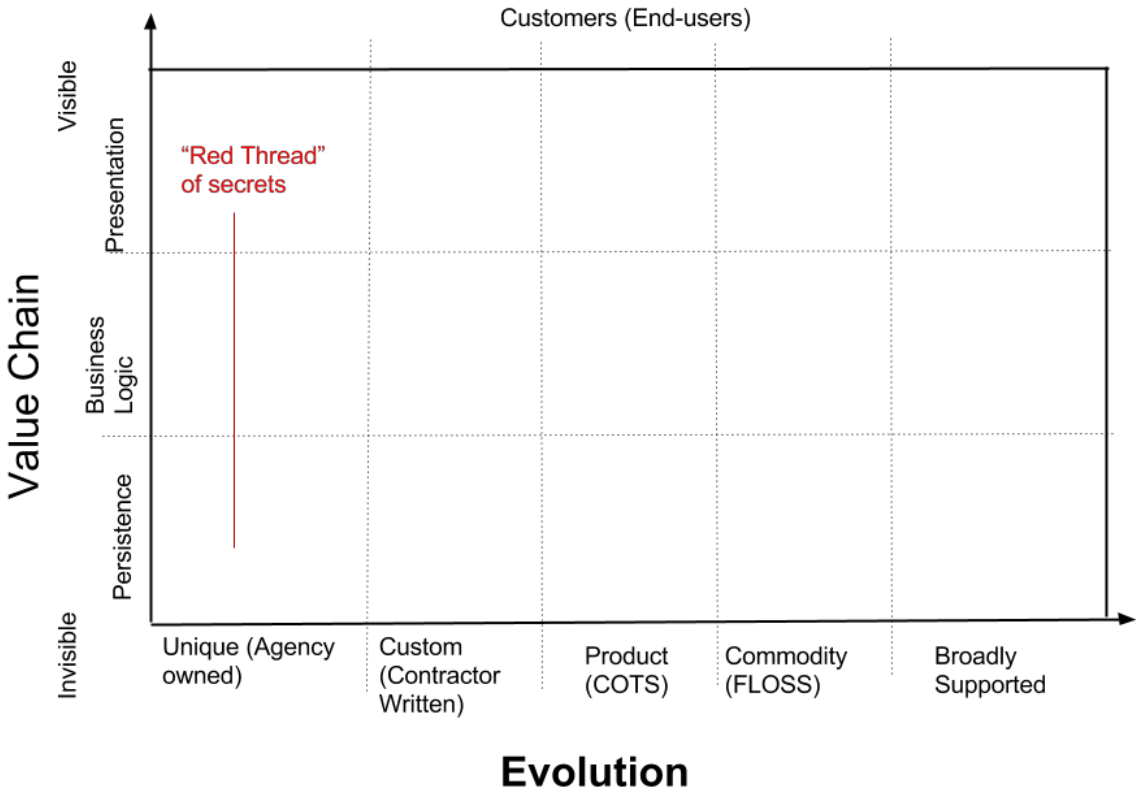
Simon Wardley has written extensively about how business people can use maps to understand a competitive landscape and succeed at business. We now recap his basic approach,

particularly aimed at the Federal program manager, acquisition officer, or executive responsible for a software system.

First, identify about the 10 or 15 most important software components of your system. Analyze them in terms of whether they are part of the Persistence, Business Logic, or Presentation layers. In messy legacy systems, a component might be smeared across several layers, which can be represented as an oval that touches several layers.

Now take each component and analyze it in terms of the Evolution axis. That is, was it written by your agency, or by a contractor? Or are you using a COTS component which is not open source? Are you using open source components, and if so, are they broadly supported or not? In many systems, you will find a mix of these categories. For example, most systems will have a small amount of custom code. Systems vary widely in terms of how many open source components they utilize.

Now draw a draft Wardley-Duncan map of our your system. A large whiteboard with post-it notes is a convenient way to do this, but you can also print out our basic disposable diagram by following this link to this diagram:



Wardley-Duncan Map Specialized for Federal Program Managers

Now that you have analyzed your system, ask yourself if any “moves” are possible. A move would be a change to your system that changed the position of one of your components. For example, can you identify a part of your system which was written by a contractor whose function has now been mostly subsumed by the increasing power and commoditization of software? Are you using a product that has an open source equivalent? In general, any move to the right on this diagram represents a chance to make your system more secure, robust, easier to support, and to save the taxpayer money.

Often you can only “make a move” by splitting a component into two or more components. For example, you might currently have a search function which combines some sort of highly application-specific filtering with a word search that could be accomplished with a full-text search engine. If you split the search component into these two separate components, one of them could be moved to the right: the full-text search engine. Unfortunately, you cannot split this node if you do not have someone technical enough to realize this possibility, and many program managers find themselves woefully understaffed of technical resources. 18F provides a technical brainstorming services to assist with this through Agile Assisted Acquisitions.

Another type of move is to move “upward” by offering new functionality which does not currently exist. For example, one of our colleagues at 18F, Sean Herron, recently helped the FDA create an open API to it’s Adverse Drug Effects database. This was the creation of a component or node at the Presentation Layer where none existed previously. The creation of GUIs for administration of relational databases and full-text search engines is another example of adding value.

Often when a new feature is asked for it can be seen as creating at least one component high in the presentation layer, close to the user, from which all value derives.

If You are Building a New System

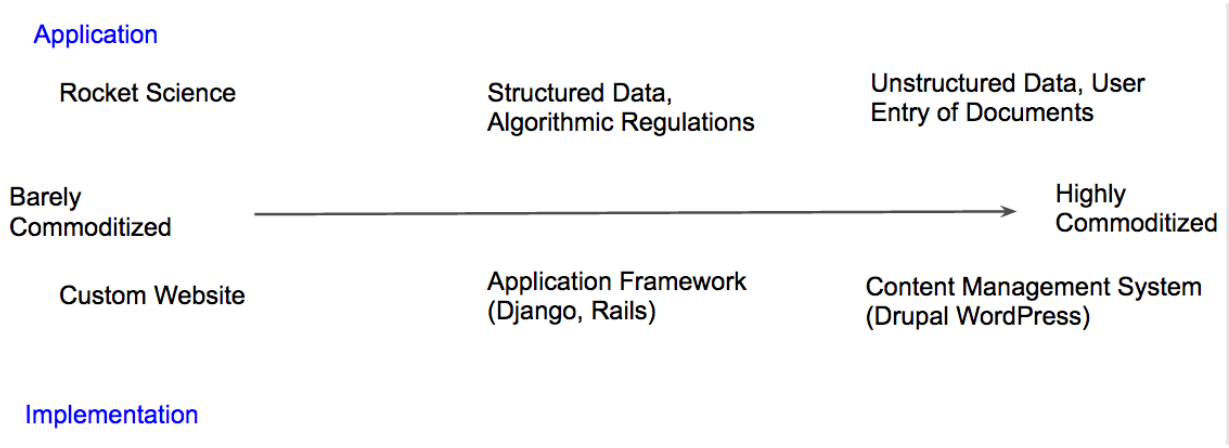
If you have the luxury of building a new, so-called “green field” system, try to keep each component of your system as highly evolved as possible. This is addressed in the US Digital Service Playbook plays numbered 5, 8, and 13. Play #9, “Deploy in a flexible hosting environment”, can be seen as recommending the same thing in deployment environment by using “cloud computing”, which is really just the commoditization of *running* software systems, as opposed to the commoditization of software *programs*.

The Commoditization of Web Sites of Unstructured Documents

Content Management Systems (CMSes) are a further example of systems which are highly commoditized in 2014. They represent the evolution of the static website of documents. By

documents we mean more-or-less unstructured data. CMSes provide search and user-addition of content in a controlled mechanism conveniently---that is, for free, out-of-the-box.

There is a spectrum of purely informational websites. Very rarely, we need some real rocket science, such as rendering a 3-D model of the Milky Way. More commonly, we need to allow search and present structured data. Very, very commonly we want to allow search and rendering of information documents. We can represent this graphically:



Do not think so much of a *either/or* choice for presenting your information, as a *yes/and* decision. Do you have unstructured documents to present? If yes, then perhaps you should use an off-the-shelf content management system. Do you also need to present highly structured records of data? If so, then perhaps you should use an application framework to conveniently present this data, in addition to your CMS. Because of these mechanisms are now so highly evolved, it may be very inexpensive to implement both a CMS and an application based on a framework, rather than forcing unstructured documents into and application framework or attempting to contort a CMS to support your highly structured data.

A Checklist of Highly Commoditized Components

Take a full, uninterrupted afternoon to apply this checklist to your system. It will help if you have drawn a fully analyzed Wardley-Duncan map of your system, or at least a good list of your components. Ask yourself how each of the following highly commoditized technologies could be applied to your project, and how you can take advantage of the commoditization of each of these to lower costs and mitigate risk by using more evolved versions:

1. The Cloud
2. Content Management Systems (Drupal, WordPress, Github Pages)
3. Full-text Search Engines (SOLR, Elasticsearch, Lucene)
4. GUI frameworks (Bootstrap)

5. Application frameworks (Rails, Django, Express)
6. No-SQL Databases (MongoDB, Redis)
7. Relational Databases (Postgres, MySQL)

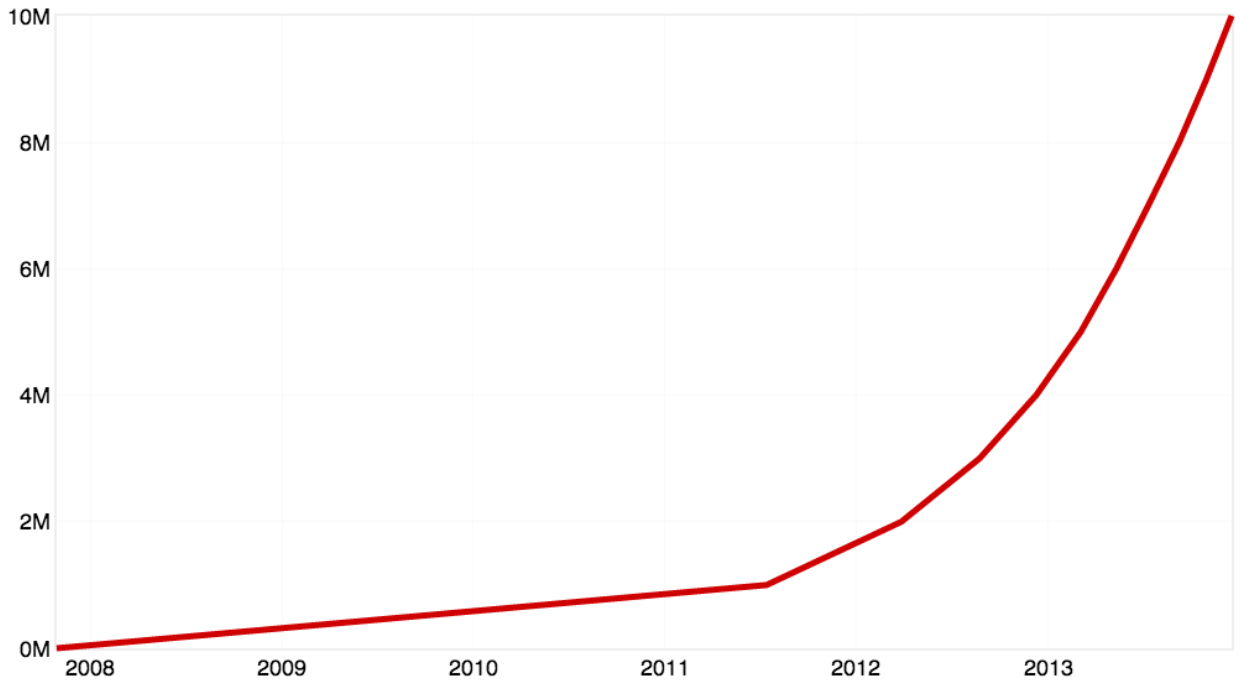
If you have a legacy system, you may very well have to imagine decomposing one of your existing components into several components. For example, if you are storing documents in your relational database today, in order to take advantage of a Content Management System you might have to split your database into two components, one of which stores structured data, and the second of which may be implemented with a Content Management System.

The Cloud is just other people's computers. For operating computer systems, the end point of commoditization is Cloud computing, or in other words, the convenient, elastically and inexpensively available computing resources in the form of complete running computers as a service. 18F is developing "Infrastructure as a Service" to make this even easier for Feds.

Using More Open Source Software

We have asserted that open source software represents the endpoint of a commoditization trend which must be reckoned with to properly serve the US Taxpayer.

The open source community and the availability of open source code appears to be growing explosively. In 2013, GitHub announced that they now had 10 million repositories, and published this chart:



However, in the words of Theodore Sturgeon defending science fiction, “90% of everything is crud.” Most of the repositories, or “repos”, at GitHub, are simple copies of more important repos. Moreover, most of the projects at GitHub do not have any great importance or an active enough community to be particularly noteworthy to the Federal knowledge worker.

Nonetheless, the tiny fraction of these 10 million systems that remain are an enormous number of valuable projects which can be used as building blocks to construct larger systems. In a sense this explosion of recombinable projects represents the fulfilment of the Unix Philosophy, which Doug McIlroy has expressed as:

> Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface.

Writing today in 2014, he might have written: “Write programs to handle JSON”, which has now become a de facto standard for web APIs.

The practice of programming at the enterprise or system level has changed since 30 years ago. Then, a programmer’s primary job was to write algorithms and data structures. Today, we spend most of our time figuring out how to reuse code that other people wrote. In this way, the Unix Philosophy, or as we more colorfully prefer to think of it, the Coming Open Source Singularity, makes us more productive.

The authors believe from their personal experience that they and other open source programmers are becoming something like 20% more productive every year, precisely because of the evolution of the open source code which we are able to reuse. This increase in productivity occurs for other programmers not working so completely with open source, but at a lower rate.

There is no more powerful force than compound interest. Twenty per-cent compounds explosively. The fact that the open source community is making each member of the community more productive so astoundingly rapidly impacts Federal program managers and Acquisition Officers highly. Each year, the cost and time to develop software is going down, and the usefulness and reliability of this software is going up.

Sadly, most of us working in the Federal technology space do not perceive it happening that way. There are two main reasons for this. The most significant is the deep outsourcing of technical expertise. The second is the cumbersomeness of acquisition practices, some of which are forced by law and regulation, and some by habit. But the use of more open source software in the Federal government can help to offset these two problems, and it is our duty to do so where possible.

How to Reuse Open Source Software

There are two senses in which we may use open source in our programs.

1. The first is to reuse open source software from established projects.
2. The second is to make the software which we write or cause to be written to be open source.

Although in principle similar, in practice these two uses require distinct considerations, and we consider the second in the next section.

Clearly, reusing existing commoditized open source software can save the taxpayer's money, even if we purchase support from a commercial firm for the software, which may be technically gratis in some modes of use. This is true whether the overall project is controlled directly by the agency, or, more commonly, the agency contracts outside firms to construct large programs.

However, open source software should not be reused willy-nilly, or to prove that it can be done, or to lend panache to the project. A project should be reused specifically to accomplish some purpose, generally to save having to write code oneself. Moreover, reusing a project represents a risk of bugs, bloat, and maintenance cost, just as writing your own code does. In general, we do not reuse a project unless it saves us about 20 lines of code.

Here are some rules of thumb to decide whether or not you should reuse a certain module, or insist that your contractors do so:

- Is the reuse saving us more than 20 lines of code?
- Is the project supported by a community whose size is proportional to the project size and complexity?
- If the project is on GitHub, look at its social activity: was it updated recently? Are there many contributors? Have many people "starred" the repository?
- Is it small enough that we can code review it ourselves, or is the community so large that we don't feel the need to directly code review it?
- Is it something which is unlikely to require frequent (more than a year) updates, and, if so, does it have a community which is producing updates in a disciplined way?

In other words, taking note both of the minor expense of reusing the project and the major expense of writing lines of code ourselves: is the use of the project saving me time, risk, and money, rather than costing me time, risk and money?

Although these considerations should be weighed, it is important to note that with experience a software engineer can make a decision about a given open source module or repository in a few minutes. Such decisions can be easily reversed if new knowledge comes to light. The burden of

deciding when to use an open-source module in your code is tiny compared to the coding time and bug fixing time that it saves you.

Our initial attempt to start a conversation about a reasonable rule of thumb for repositories might be:

> Take the number of contributors multiplied by 10 and add it to the number of times the repository has been favorited. If this number is:
>
> - Less than 100, code review it yourself or don't use it.
> - More than 100, quickly have a programmer scan all of the code files.
> - More than 1000, you don't need to code review it, but don't upgrade without reviewing the upgrade.
> - More than 1000 but less than 10,000: use it as long as it is saving you more than 20 lines of code.
> - More than 10,000: use it if it is saving you more than a few lines of code. Such a system we would call "very broadly supported", and place at the extremely evolved end of the Wardley-Duncan map.

This tentative rule of thumb applies only if the repository appears to have a relatively normal history, list of contributors and issues.

How To Use Open Source for Your Own Code

In practice, the code which your agency writes or contracts to be written indirectly must be considered separately from the reuse of code.

There are compelling security reasons that your application will be more secure if you insist that it be released to the public under an open source license from the time the first line is written, which we discuss in the Security section below.

However, even beyond security, there are several reasons to make your code open source:

1. Code which is open source by definition is not the private property or even private expertise of a single vendor. Open source prevents vendor lockin.
2. Like it or not, you are writing the legacy system of a decade from now, and the cost of maintaining it and eventually rewriting it will be much, much lower if it is written as open source from the start. Any contract written for the future maintenance or modification of the code will be based on complete knowledge of the code, which decreases uncertainty and therefore the risk to the bidder.
3. Programmers do a better job when they know that their code is essentially becoming a matter of public record. Everyone hates embarrassment, and most programmers wish to be seen as craftspersons who take pride in their work.

4. Finally, the US Taxpayer deserves to see and reuse the code that they are paying for, if they choose to do so. Even if the chances that what you write will be reused by the public is low, we should let them be the judge or that.

Benjamin Franklin was the first Civic Hacker. It is our hope that ten years from now there may be many people contributing to Federal open source projects under tight code review. Today, the public is contributing more to libraries produce by the government than to applications.

Security and Open Source

> "System security should not depend on the secrecy of the implementation or its components." -- Guide to General Server Security, National Institute of Standards and Technology

A codebase is a terrible secret.

Because a codebase is so large, it cannot easily be changed. Furthermore, it must be known, or at least knowable, to the large number of people who work on it, so it cannot be kept secret very easily. This is represented at the bottom of Figures 2 and 3. Therefore "security through obscurity" is a terrible idea when it comes to a codebase. In most cases your system will consist of code which you reuse as well as code that your write yourself. Therefore both of these types of code should be open.

Of course, your system will have secrets in most cases -- keys, passwords, and the like -- but you should assume they have been discovered and change them often. We call these secrets a "red thread", because, like a red thread in a white handkerchief, they should be as vivid and thin as possible. By making them thin, such as a single password, you make them very easy to change and keep secret. Although these secrets are tiny, they must be managed carefully and conscientiously. We believe this concept is so important that we have placed it on our reusable version of the Wardley-Duncan map linked to above.

Just as there are risks of defects and complexity associated with using open source modules indiscriminately, so too are there security risks that through negligence or by the intention of a bad actor a security vulnerability enters your system.

The key to preventing this is code review. You must make sure that each component you use is code reviewed. In practice this means either that you must use very popular projects whose code is looked at by a large number of people on a regular basis, or you must use small projects which you can code review yourself. In practice, the criteria for making this decision for reused components is similar to the rules of thumb that we have already laid down for managing risk. However, you may need to adjust these rules of thumb based on how often you plan to update the component.

For example, a small component which is very stable need not be updated at all. If it is small and you can code review it or pay your subordinates to code review it, then you may use it. On the other hand if the project has frequent updates, you will have to decide how to manage these updates. A large project may have both stable and experimental branches. In general you want to update as frequently as the major number of the branch. If the project is very active and many people are looking at it, this does not represent a security risk. If however a project is changing rapidly and producing many releases and you do not have the resources to ensure that each new release is code reviewed and you do not trust the community to do so, then you probably should not use that component.

With an open source component, it is at least possible to understand how much code review it is receiving. We know of no way to do this for closed source code kept as a secret. A firm which is asked to maintain the security of the code that it has written is placed in a conflict of interest. It is not in its short-term interest to spend resources on this code review, and it is not in its short-term interest to admit defects.

Security of Your Own Code

Make all your code open and examinable from the start. Moreover, it is best to encourage as many people to look at it, because the more people who seriously review the code the more likely a security flaw is to be found. Programmers will code more securely when there code is in the light from the beginning.

Code that you write or contract to have written should be open source from the start, because it relieves you of the terrible risk and burden of maintaining the secrecy of the codebase. This means not only that it is published under an open source license as explained in the 18F open source policy, but that it is published in a modern source code control system.

Agile Assisted Acquisitions

The principle key to this reuse is simple, but beyond the scope of this paper: A modular architecture. Because so much of the technical expertise has been outsourced from the Federal government, 18F has created Agile Assisted Acquisitions to provide the technical expertise on a consultative basis at recovery costs to guide Federal programs toward a modular architecture using Agile methodologies.

Glossary

18F	An incubator using modern software development techniques for the Federal government.

Agile Assisted Acquisitions	A consultancy of 18F to assist agencies manage IT acquisitions using Agile methodologies.
application framework	A software system that facilitates the construction of an application, such as a web application (e.g. Rails, Django, Sails.js.)
civic hacking	Programming for the public good in cooperation with government.
cloud computing	Using other people's computers.
content management system	A computer application that allows publishing, editing and modifying content, organizing, deleting as well as maintenance from a central interface.
FISMA	The Federal Information Security Management Act of 2002.
full-text search engine	A way to find things using words that occur in them.
GUI framework	A body of code which facilitates implementing a modern web-based and mobile interface (e.g. bootstrap).
Github	GitHub is a Git repository web-based hosting service which offers all of the distributed revision control and source code management (SCM).
green-field	A project unconstrained by a legacy of past work.
NoSQL database	(Not Only SQL) Data retrieval not based strictly on tables, such as key-value pair look up.
Lucene	Apache Lucene is a free open source full-text search engine.
open source	Open-source software (OSS) is computer software with its source code made available with a license in which the copyright holder provides the rights to study, change and distribute the software to anyone and for any purpose
PostgreSQL	A widely used open-source relational database management system.
relational database	A relational database stores information about both the data and how it is related based on tables.
SOLR	Solr (pronounced "solar") is an open source enterprise search platform from the Apache Lucene project.
SourceForge	A web-based source code sharing platform of many project repositories.

TechFAR	A Handbook for Procuring Services Using Agile Process
Unix Philosophy	Write small, recombinable programs.
USDS	The U.S. Digital Service which seeks to drive innovation and technica excellence.
USDS playbook	A concise guide to best practices for Federal Program Managers and Acquisition Officers.
Wardley-Duncan map	A chart of components using evolution and closeness to end-user as axes.

Further Reading

Codd, E. F. (1970). "A relational model of data for large shared data banks". *Communications of the ACM* **13** (6): 377. doi:10.1145/362384.362685. edit

Liu, Lily (2013) "When Hacking is Actually a Good Thing: The Civic Hacking Movement"
http://www.huffingtonpost.com/lily-liu/when-hacking-is-actually-_b_3697642.html

Lee, Gwanhoo, and Xia, Weidong (2010) "Toward Agile: An Integrated Analysis of Quantitative and Qualitative Field Data on Software Development Agility"
MIS Quarterly Vol. 34 No. 1, pp. 87-114/March 2010

Wardley, Simon (2013) "Basic … repeated … again …"
http://blog.gardeviance.org/2013/03/basics-repeated-again.html

Balter, Ben (2011) "Towards a More Agile Government"
http://ben.balter.com/2011/11/29/towards-a-more-agile-government/

Balter, Ben (2014) "Why Isn't All Government Open Source"
http://ben.balter.com/2014/08/03/why-isnt-all-government-software-open-source/