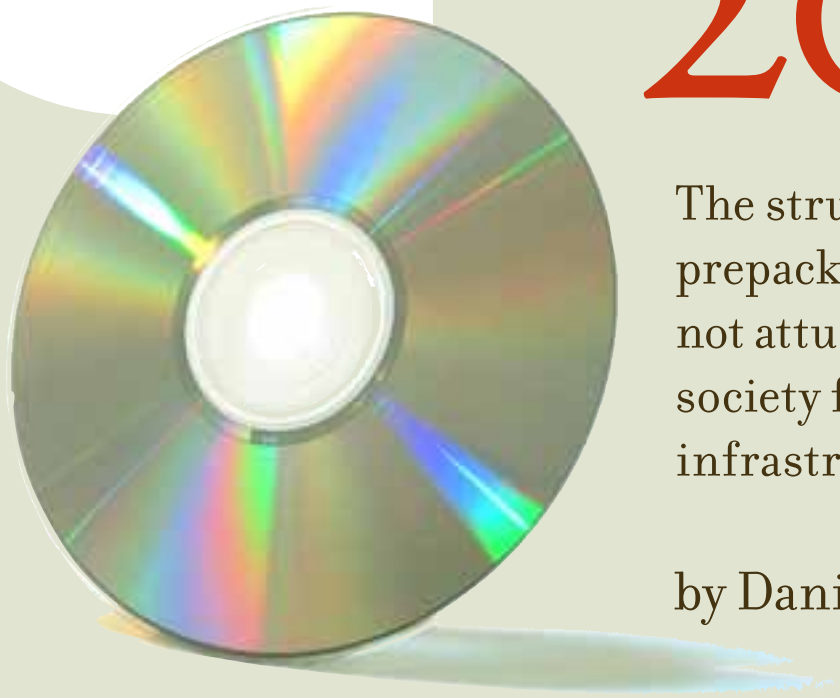
Software that lasts 200 YEARS



The structure and culture of a typical prepackaged software company is not attuned to the long-term needs of society for software that is part of its infrastructure. *continued* ▸

by Daniel Bricklin

I've been following some of the writings and actions of the Massachusetts State Executive Office for Administration and Finance as it deals with its Information Technology needs. It was through listening to Secretary Kriss and reading the writings he and other Massachusetts government officials have produced that I've come to look at software development from a whole new perspective. This essay tries to present that perspective and examine some of its implications.

MANY THINGS IN SOCIETY ARE LONG-TERM

In many human endeavors, we create infrastructure to support our lives which we then rely upon for a long period of time. We have always built shelter. Throughout most of recorded history, building or buying a home was a major starting step to growing up. This building would be maintained and used after that, often for the remainder of the builder's life span and in many instances beyond. Components would be replaced as they wore out, and the design often took the wear and tear of normal living into account. As needs changed, the house might be modified. In general, though, you thought of a house as having changes measured in decades.

Likewise, human societies also create infrastructure that are built once, then used and trusted for a long period of time. Such infrastructure includes roads, bridges, water

and power distribution systems, sewers, seaports and airports, and public recreational areas. These also would be used and maintained without major modifications after they were built, often for many decades or even centuries.

SOFTWARE HAS BEEN SHORT-TERM

By contrast, software has historically been built assuming that it will be replaced in the near future (remember the Y2K problem). Most developers observe the constant upgrading and replacement of software written before them and follow in those footsteps with their creations. In the early days of computer software, the software was intimately connected to the hardware on which it ran, and as that hardware was replaced by new, better hardware, new software was built to go with it. In the early days, many uses of computing power were new — they were the first application of software to problems that were previously done manually or not at all. The world got used to the fact that the computer version was an alternative and the special features and cost savings were what was special.

Today, hardware is capable enough that software can be written that will continue to run unmodified as hardware is changed. Computers are no longer new alternatives to other applications — they are the only alternative. Despite this, old thinking and methodologies have remained.

Computers and computer software have been viewed as being valuable for no longer than common short-term durable goods like an automobile or sometimes even tires. In accounting, common depreciation terms for software are three to five years, ten at

most. Contrast this to residential rental property which is depreciated over 27.5 years and water mains and brick walls which are depreciated over 60 years or more.

RECORDS

Another aspect of human society is the keeping of records. Common records kept by governments include property ownership, citizenship and census information, and laws. Personal records include images (such as portraits) and birth, death, and genealogical information. General information kept by society includes knowledge and expression, and artifacts representative of culture. Again, the time frame for keeping such records is measured in decades or centuries. I can go to city hall and find out the details of ownership relating to my house going back to when it was built in the late 1800s. "Family bible" records go back generations. The Boston Public Library, like many city libraries, has newspapers from over 200 years ago available on microfilm, and many from the last 150 years in the original paper form.

Most of these societal records have been kept on paper. When computers were first introduced, they were an adjunct to the "real" paper records, and paper printouts were made. Computer-readable "backups" and transaction logs were produced and stored on removable media such as magnetic tapes, or even paper printouts. These were usually written and then rarely accessed, and even then accessed in a manner akin to the newspaper stacks of a library. Only the recent, working copies of data were actually available to the computers on an instantaneous basis. Much of the use of computers was for "transactions," and only the totals at the end of the time period of interest

needed to be carried forward except in rare circumstances, such as disaster recovery or audits. Switching to a new computer system meant copying the totals and then switching the processing of new transactions to the new system instead of the old.

When it comes to moving ahead, most new software and hardware can only access the most recent or most popular old data. Old manuscripts created with old word processors, often archived on obsolete disk cartridges in obsolete backup formats, are almost impossible to retrieve, even though they are less than 25 years old. The companies that built the software and hardware are often long gone and the specifications lost. (If you are older than 30, contrast this to your own grade school compositions saved by your parents, or letters from their parents, still readable years later.)

TODAY'S WORLD AND SOCIETAL INFRASTRUCTURE SOFTWARE

The world is different now than it was even just a decade or two ago. In more and more cases, there are no paper records. People expect all information to be available at all times and for new uses, just as they expect to drive the latest vehicle over an old bridge, or fill a new high-tech water bottle from an old well's pump. Applications need to have access to all of the records, not just summaries or the most recent. Computers are involved in, or even control, all aspects of the running society, business, and much of our lives. What were once only bricks, pipes, and wires, now include silicon chips, disk drives, and software. The recent acquisition and operating cost and other ad-

vantages of computer-controlled systems over the manual, mechanical, or electrical designs of the past century and millennia have caused this switch.

We need to start thinking about software in a way more like...building bridges, dams, and sewers.

I will call this software that forms a basis on which society and individuals build and run their lives "**Societal Infrastructure Software**." This is the software that keeps our societal records, controls and monitors our physical infrastructure (from traffic lights to generating plants), and directly provides necessary non-physical aspects of society such as connectivity.

We need to start thinking about software in a way more like how we think about building bridges, dams, and sewers. What we build must last for generations without total rebuilding. This requires new thinking and new ways of organizing development. This is especially important for governments of all sizes as well as for established, ongoing businesses and institutions.

There is so much to be built and maintained. The number of applications for software is endless and continue to grow with every advance in hardware for sensors, actuators, communications, storage, and speed. Outages and switchovers are very disruptive. Having every part of society need to be upgraded on a yearly or even tri-yearly basis is not feasible. Imagine if every traffic light and city hall record of deeds and permits needed to be upgraded or "patched" like today's browsers or email programs.

Needing every application to have a self-sustaining company with long-term management is not practical. How many of the software companies of 20 years ago are still around and maintaining their original products?

SOFTWARE DEVELOPMENT CULTURE

Traditional software development falls into two general categories: Prepackaged and Custom. Prepackaged software is written by Application Software Companies (often called Independent Software Vendors, ISVs) who produce a program and then sell the same product to multiple customers. Custom software is written either by an independent company under contract or by in-house developers for a specific user. Common elements may be reused from project to project, but the overall program is unique.

Prepackaged software has the advantage of using the leverage one gets by spreading development costs over multiple users. Custom software has the advantage of being able to be tuned to very specific needs and circumstances of each user. A challenge when developing prepackaged software is developing a product that appeals to a wide audience. A challenge when developing custom software is to take advantage of "generic" prepackaged components to lower development costs.

The most successful prepackaged software applications have been those that may be inexpensively customized to meet the needs of users by developers with less and less computer skills, most desirably by the users themselves, or that form a base on which other prepackaged or custom software are built. Examples of such software are the common "productivity" applications like word processors and spreadsheets and

"plumbing" software like operating systems, database engines, and web servers. The developers of prepackaged software are driven by a need to make their products appeal to today's potential users (and buyers), usually through features that distinguish them from competition.

Funding for initial development should come from the users. The requirements for the project must be set by the users, not the developers.

A traditional prepackaged software company is organized as an ongoing enterprise, usually with a desire and plans for growth. An initial core of technical and product design people build the first version of the product. Marketing and sales people are added to sell the product and bring in revenues. Development continues and new, better versions are produced. New revenue comes from selling to existing customers, with each new version needing to give existing users a reason to replace the old product. The mentality, and the resulting major investments in corporate marketing, sales, and research activities, is focused on obsolescence and "upgrading" — but only upgrading to products from that company. The potential for new customers and upgrade revenue is often a requirement to procure initial funding.

There are prepackaged software companies that are structured to make their profits from services and activities separate from the actual delivery of software code. The

software itself may be available with no or little charge, but the organization is set up so that support of various sorts is provided by the company which has special knowledge of, and access to, the product. Again, there is a culture of obsolescence, to keep customers upgrading to new versions and paying for maintenance.

THE NEEDS OF SOCIETAL INFRASTRUCTURE SOFTWARE

Let us look at the needs for societal infrastructure software. They include the following:

- Meet the functional requirements of the task.
- Robustness and long-term stability and security.
- Transparency to determine when changes are needed and that undesired functions are not being performed.
- Verifiable trustworthiness of all three of the above.
- Ease and low cost of training for effective use.
- Ease and low cost of maintenance.
- Minimization of maintenance.
- Ease and low cost of modification.
- Ease of replacement.

- Compatibility and ease of integration with other applications.
- Long-term availability of individuals able to train, maintain, modify, determine need for changes, etc.

The structure and culture of a typical prepackaged software company is not attuned to the needs of societal infrastructure software. The "ongoing business entity" and "new version" mentality downplay the value of the needs of societal infrastructure software and are sometimes at odds.

By contrast, custom software development can be tuned better to the needs of societal infrastructure software. The mentality is more around the one-time project leaving an ongoing result, and the cost structures are sometimes such that low maintenance is encouraged. The drivers of custom software are often the eventual users themselves, paying upfront for development.

Some of the problems with custom development with regards to societal infrastructure software are the inability to spread the development and maintenance costs among a large number of customers and the narrow focus on the current requirements of the particular customer and their current stage of need (which often may change in ways visible to other customers but not yet to them).

A NEW STYLE OF DEVELOPMENT

What is needed is some hybrid combination of custom and prepackaged development that better meets the requirements of societal infrastructure software.

How should such development look? What is the "ecosystem" of entities that are needed to support it? Here are some thoughts.

Funding should also be able to come from a combination of multiple sources. Funding or cost-sharing "cooperatives" need to exist.

Funding for initial development should come from the users. Bridges and water systems are usually funded by governments, not by private entities that will run them for generations. The long-term needs of the funders must be more inline with the project requirements than the investment return needs of most private sources of capital.

The projects need to be viewed as for more than one customer. A system for tracking parking tickets is needed by many municipalities. There is little need to have a different one for each. As a result, the funding should also be able to come from a combination of multiple sources. Funding or cost-sharing "cooperatives" need to exist.

The requirements for the project must be set by the users, not the developers. The long-term aspects of the life of the results must be very explicit. Best-practices must be established, tracked, and revisited.

Development may be done by business entities are built around implementing such projects, and not around long-term upgrade revenue.

There is the whole issue of data storage and interchange standards that is critical to the long-term success and ability to do migration. Impediments such as intellectual property restrictions and "digital rights management" chokepoints must be avoided. (Lawmakers today must realize how important data interchange and migration is to the basic needs of society. They must be careful not to pollute the waters in an attempt to deal with perceived threats to a minor part of the economy.)

Another critical issue is **platform (hardware and software) independence**. All development of long-term software needs to be created with the possibility of new hardware, operating systems, and other "computer infrastructure" in mind.

The actual **development may be done by business entities which are built around implementing such projects, and not around long-term upgrade revenue**. Other entities are needed for providing the ongoing services with a mentality of keeping existing

systems running. (The two entities may or may not be related.) Many such companies already exist.

Unlike much of the discussion about open source, serendipitous volunteer labor must not be a major required element.

The **attributes of open source software need to be exploited**. This includes the transparency of the source code and the availability for modification and customization. Much has been written with regards to open source and its value for bug finding, security checking, etc., which is why this is needed. The added benefit here is that society as a whole may benefit in unforeseen ways as new applications are found for programs, be they in the private or public sector. The availability of the source code, as well as the multi-customer targeting and other aspects, enables a market for the various services needed for support, maintenance, and training as well as connected and adjunct products.

The development may be done in-house if that is appropriate, but in many cases there are legal advantages as well as structural for using independent entities. Some governmental agencies may be precluded from licensing their results under licenses that are most appropriate for the long-term health of the projects. For example, they may be required to release the program code into the public domain where it may

then be improved by others (and re-released under restrictive licenses) without a return benefit to the original funders.

Unlike much of the discussion about open source, **serendipitous volunteer labor must not be a major required element**. A very purposeful ecosystem of workers, doing their normal scheduled work, needs to be established to ensure quality, compatibility, modifications, testing, security, etc. Educational and other institutions may be employed with the appearance of volunteer labor as students and other interested parties are used, much as courts and other governmental agencies have used interns and volunteers for other activities. The health of the applications being performed by the software must not be dependent upon the hope that someone will be interested in it; like garbage collecting, sewer cleaning, and probate court judging, people must be paid.

The ecosystem of software development this envisions is different than that most common today. The details must be worked out. **Certain entities that do not now exist need to be bootstrapped and perhaps subsidized. There must be a complete ecosystem**, and as many aspects of a market economy as possible must be present.

LEARNING FROM CIVIL ENGINEERING

My friend Peter Levin pointed out to me that the analogy between software engineering and civil engineering (the building of bridges, dams, and sewers) should be used to help flesh out a potential structure of the ecosystem. Here are some more thoughts inspired by that.

Architects, civil engineers, and contractors as part of their training learn a set curriculum, pass tests, and are often licensed. They are supposed to share a body of knowledge and experience and demonstrate competence. **What thrust should be part of the training of software engineers?** For years we emphasized execution speed, memory constraints, data organization, flashy graphics, and algorithms for accomplishing this all. Curricula needs to also emphasize robustness, testing, maintainability, ease of replacement, security, and verifiability.

Like all engineering, new software, as we know, commits old errors. We need to teach the right "war stories."

Standards bodies publish best practices (how high should a railing be above the stair tread, how thick should a concrete footing be under a supporting pillar, etc.). Even though a project might be novel (such as a new bridge or Boston's Big Dig), there are many standards that can (and must) be applied. By standards here we mean a conservative approach that is intended to minimize error, increase security, and lower maintenance costs, not just facilitate data interchange. **Like all engineering, new software, as we know, commits old errors.** We need to teach the right "war stories."

Physical projects are subject to inspection by standards bodies. When you have electrical or plumbing work done, the town inspector comes to check the work before the

job can be considered finished. **Transparent societal infrastructural software needs inspection.** This will raise the role of independent testing entities. There is much talk about such roles in the discussion about electronic voting and gambling machines, but it is also important for the software we are covering here. These jobs — part QA, part auditor, part private investigator — can be very high status because of the range and depth of knowledge and experience needed. For public projects, the transparency of open source is needed to allow multiple, independent inspections. There are also different inspection specialties, including standards compliance, security and other stressing, maintainability, and functionality.

What we learn from failures enter the standards lexicon and is used for training and new design. We don't do this yet in the world of software.

When physical projects fail (a suspension bridge twists in heavy winds, an elevated freeway falls down in an earthquake, an airplane crashes) **public inquiries are performed**, reports are published, and fixes are designed and retrofitted to existing projects. What we learn from failures enter the standards lexicon and is used for training and new design. We don't do this yet in the world of software. Access to the source code, the right to discuss it in detail, and the ability to search for similar code elsewhere is crucial to many such studies.

FURTHER THOUGHTS

The heart of this is some sort of open source software. The exact license requirements are not yet clear, and will probably vary depending upon the project. The depth of thinking that went into the GNU General Public License is needed, and it is a good start.

There is nothing that says that there should only be one product for each application. Competition is very helpful for bringing out the best.

The role of **open source software scares many traditional software developers.** There is an image of a need for volunteer labor, and developers not getting paid. This is far from the case. Developers and companies are still needed, and the best will be in high demand and well paid. Criteria of what is "good" may change — the ability to write clear, robust, maintainable code with an eye to the future, or do clean modifications, or explain how to use old software in new contexts, will become even more important. Documentation, training, servicing, testing, and more will still be paid for. In fact, the knowledge that such work has long-term consequences and may be amortized over longer periods of time raises their value. What does go away is the effort spent on making upgrading and replacement a desirable thing, both in development time and marketing dollars.

What about competition? There is nothing that says that there should only be one product for each application. **Competition is very helpful** for bringing out the best in product development. With that knowledge, funders should consider funding more than one project and keep all promising ones alive even if, as is the tendency with software, one comes to dominate in deployments.

Smaller entities have a better chance than today, because in such an ecosystem they would…be evaluated …based on their products' characteristics for fitting into the ecosystem.

What does this say about the size of the development entities? There is no special requirement. Some may be very big, some may be very small. Smaller entities (and projects) have a better chance than today, because in such an ecosystem they would not be evaluated based on their own ability to provide long-term support (a major impediment today), but rather on their products' characteristics for fitting into the ecosystem.

The structure of the development may be concentrated mainly all in one entity, much as with a product like MySQL or Adobe Photoshop. Alternatively, it may instead be co-ordinated by a strong center, but distributed among many players, such as with Linux. The key skills will include the ability to manage such projects in the ecosystem. There

is probably a separation between managing the initial development and the long-term maintenance and monitoring.

The key is a model for long-term use, with a lowering of total cost of ownership, less disruption, and better integration.

Is this "socialized software," with the government making all decisions? No. While funding and management cooperatives seem a likely part of the ecosystem, there is no need for a single such entity; in fact, that would be bad. Developers with promising ideas can still use risk capital for initial development, and would still be able to find single customers to provide the funding. Also, some projects may be worth funding solely because they are synergistic with existing products that are being supported by existing entities. So, for example, a training and support company may help fund a product that lowers maintenance costs and that will need training.

Remember, **this is only for one part of the software world** — that of social infrastructure software. There are many other uses of software, each with their own preferred ecosystem for development and support.

As part of this, **buyers must get used to funding projects in advance**. This is already the case in many areas, and the addition of cooperative funding with others can lower

the costs or increase the scope of potential projects. Buyer funding lowers the requirement for potential "big hits" to incentivize development.

There is much talk about open source software in relation to existing software firms and lowering costs. What we are discussing here is **opening up new types of firms, with huge potentials for revenues** stemming from valuable services.

Open source essays often revolve around cost savings of acquisition and the use of volunteer labor for testing and maintenance. That is not the thrust here. In fact, the acquisition costs may actually be higher, and paid labor is assumed. The key is a model for long-term use, with a lowering of total cost of ownership, less disruption, and better integration. Open source discussion for government and business is often just in regards to existing open source applications, such as Linux and hoped-for desktop applications. There needs to be more discussion about projects of less general interest to the common software developer, such as EPA compliance monitoring systems, government record keeping, court workflow systems, and e-government components. Open source software discussion should be about keeping the trains running on time and not just saying it should run on Linux. The discussions should be about funding the companies needed in such an ecosystem and assuring their sources of healthy revenue. The code is not the only part of the equation, and leadership for all aspects of the ecosystem need to be addressed.

---

info

ABOUT THE AUTHOR

Daniel Bricklin, a software designer, is best known as the co-creator of VisiCalc, the first electronic spreadsheet. In late 1995, Dan founded a new company, Trellix Corporation, which became the leading provider of private-label web site publishing technology and managed hosting services.

He attended college at M.I.T., receiving a B.S. in Electrical Engineering/Computer Science in 1973. In 1977 he returned to school, this time receiving an M.B.A. from Harvard in 1979. Dan has received the IEEE Computer Society's Computer Entrepreneur Award and a Lifetime Achievement Award from the Software Publishers Association.

DOWNLOAD THIS

This manifesto is available from http://changethis.com/6.200YearSoftware

SEND THIS

Click here to pass along a copy of this manifesto to others.
http://changethis.com/6.200YearSoftware/email

SUBSCRIBE

Learn about our latest manifestos as soon as they are available. Sign up for our free newsletter and be notified by email. http://changethis.com/subscribe

info

WHAT YOU CAN DO

You are given the unlimited right to print this manifesto and to distribute it electronically (via email, your website, or any other means). You can print out pages and put them in your favorite coffee shop's windows or your doctor's waiting room. You can transcribe the author's words onto the sidewalk, or you can hand out copies to everyone you meet. You may not alter this manifesto in any way, though, and you may not charge for it.

NAVIGATION & USER TIPS

Move around this manifesto by using your keyboard arrow keys or click on the right arrow (→) for the next page and the left arrow (←). To send this by email, just click on ✉ .

KEYBOARD SHORTCUTS	PC	MAC
Zoom in (Larger view)	[CTL] [+]	[⌘] [+]
Zoom out	[CTL] [–]	[⌘] [–]
Full screen/Normal screen view	[CTL] [L]	[⌘] [L]

BORN ON DATE

This document was created on 18 October 2004 and is based on the best information available at that time. To check for updates, please click here to visit http://changethis.com/6.200YearSoftware

info

ABOUT CHANGETHIS

ChangeThis is a vehicle, not a publisher. We make it easy for big ideas to spread. While the authors we work with are responsible for their own work, they don't necessarily agree with everything available in ChangeThis format. But you knew that already.