developer.*
The Independent Source for Software Professionals

Stunted Growth
Subsidies and Stagnation in the Software Tools Market

By Steve Benz

Recently, I broke my wrist. I went to a minor emergency clinic, where they took an X-ray. This machine, a standard film device, was probably 20 years old by the look of it. The doctor examined the images and sent me on to a specialist. The specialist put my wrist in a cast and ordered a new set of X-rays to confirm that the bones were properly aligned after casting.

The next X-ray machine was a modern unit, one that produces a real-time image on a monitor, allowing the technician to perfectly position the joint. If there is some slight movement that blurs the image, the technician can see this right away and snap another shot. I commented upon how marvelous this machine was, and he said, “I don’t know how I could do my job without it.”

When economists comment on what caused the boom of the 1990’s and the expanding economy of today, one word you will surely hear is *productivity*, and this newfangled X-Ray machine is a perfect example of what they are talking about. With this new tool, the X-Ray technician is able to produce a far better result in less time; because of it, the technician is able to demand a competitive salary.

When you stop and think about it, you can probably list off a few devices like this X-Ray machine yourself. What’s striking is that most of the “can’t do without it” machines have *software* as a primary component.

The “can’t do without it” machines can’t do without us!

This fact makes it all the more perplexing that *we* don’t have a “can’t do without it” machine for software development—which is not to say that there has not been any improvement in software tools and practices. I just mean that those advances that have happened are not as great as workers in other industries have enjoyed. The sad fact is, we have been spending our creative energies helping others do their jobs, but few of us have been able to do anything for our own productivity. This has dire consequences for our future salaries, the future of our industry, and the future of the global economy.

If development tools remain stagnant, salaries for software developers will not keep pace with the economy as a whole. The best young minds will choose other majors at Universities and the resulting dearth of talent in the software field will then slow production of productivity-enhancing machines for other industries, thus causing the whole economy to slow.

This article explores the question of why development tools have not advanced much in the past couple decades. Other authors have commented on this, and most emphasize the difficulty of creating development tools. I do not dispute that, but would point out that if people are given sufficient incentive they can overcome most any challenge. I intend to show that persistent economic conditions are the root cause of the slow growth in the development tools industry.

Improvements in Coding Productivity

When I first began pondering this dilemma, I did what I usually do when presented with a terrible conundrum: I went to lunch with some of my clever friends. Perhaps they could convince me I was blind and that productivity has, in fact, been improving and I just hadn't noticed.

The first candidate for a "can't do without it" machine was the IDE, but the Emacs contingent quickly refuted that; they showed that IDE's brought together a set of tools (editor, debugger, browser) that were already available. The IDE is easier to learn, but beyond the learning curve an IDE is not necessarily better to work with than a well configured set of separate tools.

That said, continuous compilation is one aspect of the IDE that seems like it may become a bona-fide "can't do without it" technology. Code completion can significantly improve coding speed, and the "refactor" capability allows us to make changes we might not have even attempted in earlier days. Still, we all felt that the current capabilities of IDE's made us a few percent more efficient, but not an order of magnitude.

C++ and Object-Oriented Programming have come into widespread use, but we all noted that people had been bodging C and other languages to do class-like things before the formal introduction of C++, so you could say that it really isn't all that new. Templates are new, but they still have yet to realize their potential in C++ because many engineers find them too difficult to use. Their counterparts in Java and C# are still too far out on the leading edge for most shops to be able to use them. Once they do, there's some likelihood that the same thing that happened to C++ templates will happen to Java and C#.

UML and modeling systems have gained traction, but still have not gained anything like universal acceptance. Further, the UML modeling paradigm is geared toward helping improve the productivity of a *team* of people with a mixture of specialties and concerns as opposed to helping an individual programmer. So, in a sense, UML is more of a tool for software product managers than a tool for software engineers.

In the end, we were forced to conclude that a programmer using tools that were state of the art circa 1990 was indeed only marginally less effective than the same programmer using the tools we have today.

What is Stunting the Growth of Development Tools?

This regrettable lack of progress is not new; Frederick Brooks observed it back in 1986 in his essay, "No Silver Bullet." [Brooks, 1987, 1995] In this essay Brooks argues that we are not likely to see a single innovation that increases programmer productivity by an order of magnitude or more, as some past innovations have. In this respect, history has proven Brooks correct.

What Brooks did *not* foresee in this essay were the hundreds of small ideas that have crept into practice since he wrote his paper nearly 20 years ago. The aggregate effect of all of these has had a meaningful impact on productivity, certainly more than any of the large movements that were going on back then. So when we ask why productivity is growing so slowly, we should look away from the big schemes and ask why the small ideas are so few in number.

So, why is it that with all of us staring the problem of our own productivity in the face day after day that we have not come up with a "can't do without" technology to help ourselves?

In order to make a new product happen, you need three things: a good idea, skilled people, and the motivation to try it. We know that the skilled people are out there. Moreover, we know that these skilled people don't seem to have any problem coming up with ideas to help other industries, so we should assume that they are also coming up with ideas in the software development space. So it must be the motivation that's missing.

Most software developers just love writing a good tool, and many of us will write them just for the satisfaction alone. But when it comes to selling a product, you need more than just the product. You need sales people, documentation people, test engineers—and you can't expect all of them to share your zeal; they will want to get paid. Software tool development, then, is just like any other industry: “motivation to do it” mostly comes down to money.

On the face of it, there should be a great deal of money in the software tools business. There are lots of people out there getting paid handsome salaries to develop software. But, surprisingly, the money really isn’t there for small businesses. This article will show that the ultimate cause of this deficiency is the fact that most of the large development tools are subsidized by the sales of Operating Systems and hardware. These subsidies have the effect of diminishing the profit potential of any pure software tool vendor and thus remove the incentive to create.

Following the Money

The 1980’s and 90’s were abuzz with the term *software crisis,* a phrase coined when it first became obvious that we were having a tough time coming up with software that was as good as the hardware, and that there was not enough predictability and productivity in the software creation process. It was generally believed that since the demand for software was so great, anyone who could come up with a tool for doing “Rapid Application Development” would be in a position to make a nearly infinite amount of money, because once this better mousetrap was invented, software shops would be obliged to adopt it or go out of business.

During those years, a great deal of venture capital money was poured into the development of software development tools—to very little effect, unfortunately. The few survivors of this era (e.g. Rational Software, now a part of IBM) live mostly in the very top end of the market—that is, they sell expensive software to large organizations. In order to sell into this sort of market, most of their efforts are devoted to making a *team* of programmers more effective, not towards making *each* programmer more effective. Thus the survivors of this era largely quit making development tools and started making collaboration tools.

The great multitude did not survive. That’s a shame, because they had a number of worthy ideas. Atherton Technologies, for instance, created and touted the “Atherton Backplane,” which was not an IDE, as such, but rather a platform onto which CASE tools and other software engineering tools could be built. This is the same core concept at the heart of Eclipse.

As a separate product marketed by Atherton, devoid of any agenda other than making money, this idea faltered. When IBM undertook the same idea, it succeeded because IBM's motivation is not to make money with Eclipse or WebSphere, but really to promote its platform and enhance its other business units' places in the market.

Vendor-Supplied Development Tools

IBM's Eclipse is an example of a *platform-vendor-supplied* development tool. That is, it is a development tool built to help the vendor promote and exert control over a platform. In this particular case, IBM created Eclipse to help it sell its web servers and consulting services. Microsoft is another example: they produce Visual Studio and the rest of their line with the intent to maximize the success of the Windows operating system line. Similarly, Sun produces their Java compiler and related tools with the intent of selling more server hardware. In fact, most companies that have ever offered a successful operating system also offered a development toolkit.

The crucial thing to understand about platform-vendor-supplied development tools is that these products are not required to be profitable. The platform vendors see development tools as a means to increase the sales of their platform, their other software lines, consulting services, and for some, hardware as well. To that end, all of them introduce things into their toolkits to make it difficult to port software written with their tools to other platforms. Of course, they all walk a fine line here; if their tools are branded with the epithet of "non-standard" then customers might stray to alternative development tool vendors.

Contrary to what the marketing literature would have you believe, vendor-supplied development tools are not a hotbed of innovation. Their goal is to be firmly in the mainstream, to attract the broadest spectrum of developers. Should a new technology evolve elsewhere that proves to be popular, it will be incorporated into the platform vendor's product. If you want to talk to a Product Manager at one of these companies about a new idea for an extension to their environment, their response always comes down to, "I'll think about incorporating it when a lot of my user base asks for it." These vendors do not promulgate tools and techniques; they consume them.

Turbo Pascal and the Consequences of Complacency

Conservatism, taken to extreme, leads to revolution. This is true in government, economics, and certainly the software tools industry. Just as the platform vendors have learned to be cautious about introducing untried technologies into their products, they also need to absorb innovative technologies into their product. If they don't, they again risk losing control over their platform.

Let's look at a historical example of this effect, the introduction of Turbo Pascal, which is arguably the most significant development tool ever released. To set the scene, in 1983 the PC was very new, and the compilers offered by Microsoft, IBM, and others were stunningly bad. They were large, old programs, some ported from mainframes and force-fit onto the PC platform—and they were quite expensive.

The PC knocked down the antiquated idea that computers were solely institutional commodities. However, development environments were still being bought and sold as they had been before the PC. Although IBM is credited with "inventing" the PC, they built it out of pre-existing parts. It was how these parts were packaged and marketed that made the PC succeed where CP/M machines had not. Turbo Pascal was a repeat of that phenomenon. The idea of the IDE wasn't new; it was the packaging of it on the PC platform, marketing it to individual developers, and then pricing it at one-tenth the cost of competitors that made the magic happen.

Turbo Pascal was ridiculously successful, so successful that Microsoft was obliged to pull its Pascal offering from the market altogether. It's hard to measure exactly how much of a problem this caused for Microsoft, but one thing is certain: they didn't like it very much.

Turbo Pascal made Borland into the dominant force in the PC developer tool marketplace, but Borland ultimately lost control of the market. Microsoft regained the market by traditional means: they spent a lot of money on development and then worked diligently at making sales. Any company with sufficiently deep pockets could have supplanted Borland at that time had they done the same thing, but it's instructive to note that Microsoft *did* while other companies (e.g. IBM) did not. Control of the DOS and Windows platforms was simply more important to Microsoft than it was to anybody else.

At the present, Microsoft is in a position to either spend Borland completely out of existence or buy them outright. And yet they do not. In any normal business proposition, they certainly would, so that they could drive up the prices and make some money off of their considerable investment in controlling the market. However, as they are not terribly concerned about making a profit on their development tools, Microsoft behaves differently. Their main concern is to avoid both being too aggressive and being too conservative. The presence of a vendor like Borland allows Microsoft to gauge new technologies and to keep its technologies modern without ever appearing to be too far on the edge.

But the principal lesson to take from the Turbo Pascal event is this: platform vendors *must* incorporate ideas from the community into their products. While this is a good thing for the platform's development community, it's a very bad thing for the small tool vendor, because there's no rule at that says the tool vendor has to get paid.

Web Services and the Struggle Between C# and Java

"Clash of the Titans" struggles like Microsoft vs. Borland are fairly rare, but we happen to have another one going on right now between Microsoft and Sun. When trying to understand where this struggle will go, it's instructive to note how it started. Microsoft's initial reaction to Java was to try to out-Java Sun by creating the Microsoft JVM. They employed the long established technique of introducing subtle incompatibilities between their Java and Sun's Java, so that if you developed for the Microsoft platform, you would have a difficult time moving to Sun's.

By the time Sun made that plan intractable, it had become clear that web servers collectively were not extensions to an operating system platform, but rather a platform in and of itself. When Microsoft was forced to conclude that they could not co-opt this new platform from Sun, they set about creating their own platform.

At the outset, Microsoft knew that they had a very safe market base. The developers who have been working on other applications using Windows platforms will want to stick with a Windows platform for developing web applications. Just as we previously discussed, Microsoft had only two ways to lose their market: either appear to be out-of-date or appear to be too far out on the leading edge.

So when Microsoft designed C# and the other .Net languages, they started with the core concepts from Java and added just a few carefully chosen refinements. Most of the refinements they chose were openly discussed as refinements to Java well before C#. So even though Microsoft created a whole new language, they chose a very conservative path.

Going forward, we can expect to see incremental improvements to both the Java and C# toolkits, executed virtually simultaneously as both try to “me too!” one another. Unfortunately, even though a huge amount of blood and treasure will be undoubtedly be expended, we really shouldn’t hold our breath waiting for something dramatic to emerge.

Big Money, Small Result

We have shown that the platform vendors are the dominant force in the development tools market, but also that they are the most conservative. Moreover, even if they wanted to become more aggressive, they could not because they can only employ a tiny subset of the people who have good ideas about software tools. Consequently, their offerings primarily consist of things that have been conceived, implemented, and proven useful by others.

This is not unique to the software market. Generally speaking, if you are looking for real innovation, you don’t look at large, well-funded labs. You look in garages and basements. This isn’t because the garages and basements are better places to do research, it’s because there are a whole lot more garages and basements than there are well-funded labs.

The Low End of the Market

Let’s say that you have written a nifty little program that simplifies your daily grind a little bit. Let’s further suppose that you find that quite a few other folks would benefit from your program. What are your options? In many cases, your hands are tied because you wrote the tool on the company’s dime, and they deny your request to publish it. But let’s take an optimistic case and say that you own the code. What then?

Publishing commercial software is not easy, and definitely not free. Still, you can make a decent living selling a software development tool. The problem is that you face the constant risk that a platform vendor will release a new version of its IDE with your tool’s functionality built in, which could destroy your business entirely.

Given this reality, it’s little wonder that people commonly choose to release a product as some form of freeware. Free software enjoys a number of competitive advantages aside from being free. The trouble with the freeware approach is that the author’s initial vision is often not quite enough. That is, the author designed the software under the assumption that it would be used in a particular environment, and in certain ways. The wider world includes diverse environments, and not all people will have the same notion of how to use the tool as the author did.

Inevitably, your product will require some support. Ideally, your users will take some of the load off of you, but you really can't count on that. In most cases, only a tiny fraction of freeware users will contribute to the code. The reality is that even open source code requires sales, marketing, and support; the author must supply most of this. If you falter, your tool will likely go nowhere.

But let's assume that your tool gains some traction, and you continue to maintain it. Further, let's say one of your users looks at it and thinks of an idea that substantially improves on what you have done. If this person wants to charge money for his tool, he now has even more problems than the ones you faced originally, because he has to compete with your tool.

Perl's Unrealized Potential

Even one of the most successful freeware software tools ever, Perl, has unrealized potential. Perl is fantastically popular, but the tools surrounding this language are less mature compared than those available for other, far younger languages (e.g. C# and Java). Perl has solved a large number of problems and has increased the productivity of countless people. It's hard to say how much, but even the most conservative guess as to the net productivity gain from the creation of Perl would be in the hundreds of millions of dollars—and yet its principal author has had to maintain a day job to support his family.

Perl and Turbo Pascal make an excellent comparison. While both products have been hugely influential, neither was really new in and of itself at the time of creation; each is an amalgam of existing ideas, put into an attractive package and sold at a very low price. Turbo Pascal fused the notion of an IDE with an existing, already popular language and a fast, lean compiler. It's unlikely that Turbo Pascal would have succeeded in the way it did had Borland not substantially undercut its competitors on price. And even though the price was low, it was still enough for Borland to pay a team of developers to make Turbo Pascal better.

Perl fused a bunch of disparate and often incompatible shell scripting languages and tools into a unified, platform-neutral language. Due to the culture and economics that existed at the time, one could argue that Perl could not have succeeded had it not been free. It's also reasonable to say that Perl would not have become the *de facto* language of CGI had it not been free. There is no doubt that Perl is a good product, but imagine what it could have been if it had the kind of revenue stream that Turbo Pascal enjoyed in its time.

Solutions

Unwinding the effects of subsidies is tricky, as their consequences are not always entirely obvious. On the surface, a subsidy of any form is just free money and can do nothing but good, but really, it's not so. One form of subsidy whose deleterious effects are starting to become well known is food aid to developing countries. Agricultural subsidies in the developed world are a long established tradition. These subsidies keep our food prices low, our farmers paid, and their lobbyists very well paid. Thus, they are popular.

It seems obvious that it would be helpful to ship agricultural surpluses created by our subsidy programs to places in the developing world where there is hunger. However, developing countries usually have an economy based on agriculture and they certainly do not have the resources to subsidize their own farmers. These countries became impoverished partly because world food prices are artificially depressed by subsidies in the developed world. When boatloads of free food arrive, the local farmers have little choice but to find some other way to make a living.

The money that the platform vendors pour into their development tools is entirely analogous. The influx of money decreases the overall price of development tools by a small margin, but it severely reduces the profitability of unsubsidized, independent vendors. Free software equates to the donated food, and the independent software tool vendor is the unfortunate farmer who gets squeezed out and has to find something else to do.

The sad fact is, software developers have been spending their creative energies helping others do their jobs, but few are able to do anything for their own productivity. This is because the development tool market is doubly cursed. First, we have the platform vendors capping the amount of money that can be made in the business. This causes a great deal of software that should be sold commercially to be given away freely. And second, the free software further limits the amount of money that can be made from software tools.

The net result is that most good ideas in this software development tools space have no commercial outlet. The inventor is left with little alternative but to either table the idea altogether, or to release it as freeware and hope to gain some little fame out of it. Ideas that would in any other market sector be proudly paraded in front of drooling investors are, in the software development space, either dropped altogether or languish for want of somebody to work on them.

Just as those in the developed world do not intend for their agricultural policies to hurt other countries, the platform vendors and other large players do not mean to hurt the small tool vendor. On the contrary, they all see the advantages the small players can offer and so have individuals and even entire departments dedicated to helping these shops with sales and marketing. The problem with this is that it starts too late in the process. Since there is such limited upside to creating a development tool, few of them ever get to the point where marketing is even an issue. The platform vendors give with one hand and take away with another.

Change can only come when the large players recognize that they are choking off the very innovations that can drive sales of their platforms. The programs they have in place to assist small tools companies are admirable, but they are also entirely insufficient. They need to set up a system to help ideas in their infancy. Further, should one of these nurtured ideas become popular enough to be incorporated into the vendor's main development tool line, the author should be fairly compensated.

If history teaches us anything, it is that there probably will never be a "Eureka!" moment where somebody discovers some single thing that can instantly multiply productivity across the board. Innovation comes in small packages from unexpected sources; most of the apparently dramatic changes we see are just several small ideas knitted together. Right now, most of these small ideas die silently because they can't make money. Somehow, we need to change the underlying economics of the development tool marketplace so that it fosters innovation, rather than stifles it.

As we have shown, it is now and will probably remain very difficult to ever reap large rewards by creating a popular software tool. In the end, the platform vendors will copy any idea that's good enough to become popular and price it such that it will not be profitable. This is an effect that probably cannot be changed.

So, if people cannot be persuaded to bring their tools to market with the promise of a big reward, we need to make it so that they can bring their products to market with substantially less financial risk. The best way to do that would be to have the platform vendors provide guaranteed loans and outright cash grants to individual developers and small shops that want to produce a software development tool targeted at their platform.

Perhaps there are other ways in which the problem can be attacked, but in any case, we simply must do something to break the established trends. The slow growth of productivity of the programming market has already led to decreased wages. As a programmer, this is of grave concern to me.

However, more than just us programmers should be concerned. As we have shown, this computer-driven rise in global productivity is something we developers are responsible for. It's not only our salaries that are in jeopardy, but also that of the X-Ray technician and everybody else who touches a computer in their daily work.

Conclusion

In the past, it has been widely assumed that the only way to get more productivity out of software development teams is to make them work together better. In fact, when we look at the definition of the software crisis, it is couched in just those terms. This viewpoint ignores the fact that team performance can be improved by improving the performance of the individual team members.

The main reason for this one-sided thinking is the belief, propagated by Brooks and others, that there is little that can be done in this arena. As proof, they point to the long history of anemic progress in development tools and techniques. In this paper, I have tried to show that the slow progress is not due to the inherent difficulty of the problem, but rather due to economic conditions that have persisted in the marketplace since its birth.

Declining salaries and declining university enrollment are signs that we are on the verge of another crisis, and this time, it is not about the team, it's about individual performers. If we are to do anything about this new "software crisis," it will have to start with a realization on the part of vendors like Sun, Microsoft, and IBM that their policies are hurting themselves, the software industry, and the entire global economy.

###

References

[Brooks, 1987, 1995] Brooks, Frederick P., "No Silver Bullet: Essence and Accidents of Software Engineering," Computer, Vol. 20, No. 4 (April 1987) pp. 10-19; reprinted in *The Mythical Man-Month: Essays on Software Engineering, 20th Anniversary Edition*, Addison-Wesley, 1995.

###

Steve Benz owns Tall Tree Software company (`www.tall-tree.com`), which makes a documentation tool called DocJet. Steve also does consulting work in everything from web applications to high-availability solutions.