Better Development Tools Are Coming, Will They Be Good Enough?

John Clark

Abstract:

It's becoming increasingly apparent that there is major room for improvement in software development tools. New and improved development environments seem to be coming out for the Macintosh almost monthly. This paper discusses known techniques for making software easier to develop and reviews the major products coming to market. It also reviews and amplifies the reasons we need better tools and the costs of developing with poor quality tools. Existing strategies offer partial solutions to the problem. A proposal is made for an alternative approach that allows for not only known techniques but also expected future techniques. These can be incorporated into a single system that does not require starting over from scratch when techniques are added.

I. Why better tools are urgently needed

If you do software development, you may have found some serious shortcomings in your development tools. You may have wished your development environment was faster or that it worked better at solving the problems you actually encounter. You may have even seen situations where the tools get in the way more than they help.

Ok, it's pretty obvious that we need better tools, but why urgently?

The costs of using less than the best possible tools

The most obvious cost is productivity. Here the cost shows up in extra personnel needed to do the job. But this cost is higher than one might first expect. It's well know that doubling the staff does not cut the development time in half. You need something more than twice the people because bigger groups need more time to stay coordinated and because there are more problems in communicating.

But reduced productivity is not the only cost, maybe not even the most serious cost. Poor tools increase the number of product quality problems. Why? Partially because it takes longer to find and fix problems with available tools (which schedules don't always accommodate). It also happens because some problems are nearly impossible to find without the right tools.

Innovation also suffers. It's much easier for people to be creative when they can try ideas quickly and painlessly and they don't have to try to justify large resources to developing new and yet unproved ideas. Poor tools don't prevent innovation but they raise artificial barriers to it and make it much harder to achieve.

Another high cost of poor tools is that it can seriously impede your organization's ability to achieve its goals. If you use them to develop your products, it limits the scope of your products and increases development time. Your product plans may not be realized or may be realized at less than their full potential. If you rely on computers to manage and run your business, it makes it much more difficult to get the information you need and the kind of tools your people need to run your organization.

For the development team, using poor tools muddles the thought process, rather than helping it. It forces them to concentrate on dealing with details of the tools rather than the real problem.

How much improvement can really be made?

That's a very good question. I've been unable to find reliable estimates of even the productivity side of the costs. Plus it depends on how much improvement can be made in tools and how much of your particular project is actually affected by the tools.

In any case, it's not too hard to justify that a two-to-one improvement is possible.

That is, better tools could cut the time and cost of development in half. In many situations, and with the right tools, it could be as high as ten to one. Meaning it costs ten times too much right now.

And we haven't even considered the other costs.

Why hasn't more progress been made?

If these figures are even remotely correct, why hasn't more been done about it?

For one thing, I don't think many people understand the magnitude of the problem. It seems inconceivable that we may be wasting over half of our time and money due to inadequate tools. Inconceivable being the operative word. We just don't picture how big this problem is. Since the kind of tools we need aren't available yet, we can't exactly "A/B" test them against what we have now.

We tend to live with the tools we have. After all, most of us were hired to develop a specific product, not to do tool development. We live with relatively short term deadlines and don't have time or money to make better tools ourselves. Strangely enough, I think this is true even for most tools developers.

It's easy to think that we already have tools to do development with, why do we need better ones? Why should we waste time getting better ones? Why can't we just use the ones we have now?

We assume someone else will do it. Somewhere an entrepreneur, academic type or big company will come up with "the answer." Or we just have to wait for better tools to evolve. Either way, we feel it's someone else's job.

One of the reasons I've heard given is that there isn't enough money to be made in development tools. The market is too small. This is especially disturbing since I've mainly heard it from programming tools developers.

The "one-track mind" problem

New tools usually come about when someone gets frustrated with the limitations of their old tools. They come up with a better way and get inspired to go to a considerable amount of trouble to create a new tool.

So what's wrong with that? The problem is that, with all the energy focused on the new ideas, they tend to stop looking for other ideas that could also help.

This wouldn't be so bad except that there usually isn't a good way to incorporate other ideas later. Even the most open systems, such as MCL[1] and Smalltalk, are easy to modify in some ways, but very difficult to modify in others.

Today, the big thing that many people narrowly focus on is object-oriented programming (OOP). But OOP alone is not enough. As an example, if you have painfully slow OOP, then you still have painfully slow programming.

The largest example of this narrow focus may be Taligent, the joint venture between Apple and IBM. Taligent has devoted huge amounts of resources on a project which has a single predominant theme, being object-oriented. I use the word "may" because there may be hope. Taligent does make references to addressing the broad problem of easing the programming process. Having an open system also seems to be high on their priorities.

Are significantly better tools really possible?

Yes! In the next section I list a number of techniques for making development easier. All of these have either already been implemented somewhere or are not technically difficult. So there is little reason to doubt that it can be done.

[1] MCL-Macintosh Common Lisp

II. Techniques for making software easier to develop

A. Turnaround time or responsiveness[2]

Responsiveness should not be thought of as just a “nice” thing to have. Not merely luxury you can afford only if you get it for very little effort. If turnaround is slow, the feeling of feedback the user (programmer) gets is greatly diminished. Instead of a tool that seems like it does your bidding, it becomes a process you set in motion that responds in its own time.

[2]Although this is a programmer interface issue, it is so important that it should be treated separately.

With more significant delays, the user naturally starts thinking of other things. When he finally gets a response, he has to put away his new thoughts and try to recall where he was. This becomes the human interrupt mode. Unfortunately, humans do not handle interrupts nearly as well as computers. Interruptions cause both a significant lag time in recovering from the interruption and a considerable decrease in productivity under interrupt conditions.

I've classified responsiveness into four categories. These are purely empirical and the times mentioned were picked from my "gut feeling." No doubt someone has studied this in detail and can give better values for the time ranges. I have not yet had time to look for studies on this.

Levels:

•A. Immediate - less than roughly 200 ms (0.2 seconds)

It feels like your action directly produces the result. The user feels his action is directly connected to the result.

•B. Sluggish - roughly 200 ms to 2 seconds

While you can still connect your action to the results, the system feels sluggish, as if you only partially have control of it.

•C. Delayed - roughly 2 seconds to 20 seconds

You're no longer sure you are really under control. Eventually you learn that you do have control, but initially you have to take it somewhat on faith.

Imagine driving a car that starts turning 5 seconds after you move the wheel. Would you drive such a car? You certainly couldn't drive it very fast.

•D. Interruption mode - roughly greater than 20 seconds

While waiting, you start to think about other things. Getting back to what you were doing takes the effort I described above.

These levels are easy to verify. Compare opening a small application that opens quickly to one that takes several seconds to open. Doesn't it feel like you have more control with the small application? Compare even the small application to switching to another application under MultiFinder. This time your action really seems connected to the result.

Why are the differences listed under the "Interrupted" mode significant? I can't prove it, but I think it affects the speed of learning. You tend to learn to use the quickly responding actions in less time than the delayed ones. There is also much more of a sense of enjoyment when the computer does what you ask nearly immediately. Delay makes you feel frustrated, impatient with the stupid machine.

Development systems (and applications) should strive to meet "Immediate" (A) levels of responsiveness. Is it unreasonable to expect response in less than 200 milliseconds? Consider this. Even a moderate speed 68030 should easily execute over 100,000 instructions in 200 milliseconds. Unless we have huge amounts of data to process (such as pictures) or we HAVE to wait for disk I/O, an incredible amount of work can be done in 100,000 instructions. So, I feel that a goal to having all but disk intensive operations occur in less than 200 ms is attainable. Even the disk I/O problem can be sped up greatly if done properly.

B. Programmer interface issues

1. Text based vs. graphic and other representations

We are still, for the most part, using text based programming. The obvious problem with it being text based is that it requires typing. Typing is not only a lot of work but it's easy to make errors while doing it.

We make identifiers[3] long so they can be meaningful, which means both lots of typing and lots of chances to make mistakes.

There are other ways of representing code. One

[3]Identifiers are how programs locate either specific pieces of information or other parts of the program.

is to use diagrams. In tools such as Prograph and Serius, coding is done by drawing connections between objects. These objects represent program or data elements and the lines that connect them represent program or data flow.

Another alternative is to treat the code as an outline. You can then hide details that are not currently relevant, which are represented by lower levels of the outline structure. While you still end up with a

text based system, you at least get some control over the amount of complexity you are presented with.

If these alternative representations are used as other ways of looking at the same information, you have a new type of tool. Essentially, these are multiple views into the same program. For example, you could use a class hierarchy diagram to help understand the relationships of your classes.

2. Coding should be a last resort

One good way for improving programming productivity is to only program (code) when absolutely necessary. It's much more time consuming to write code for things that can be drawn or otherwise designed directly. For this to work well it must be done is such a way that it meshes cleanly with the code. You can't afford to redo the coding whenever you change the non-code parts.

3. Integrated CASE[4] tools

Today's CASE tools work by reading source code and determining the same things the compiler already does. If the development environment made this information available to outside tools then CASE tools vendors would not have to re-invent this part of the world.

With the right kind of development environment the CASE tools could work both ways: analyzing the code to make it clearer, and allowing the programmer to produce changes from that perspective.

4. Object-oriented programming[5](OOP)

Object-oriented programming is one of the best tools we currently have for getting some control over the complexity of today's development environments. (Please remember that it is NOT the only one.)

Also, though we tend to think of it as a means of programming, it is also a way of organizing our thinking and structuring our ideas.

Simplifying the development process and code reuse are frequently touted as being the big virtues of OOP. From what I've seen, with OOP you gain by having somewhat simpler code, but you have to pay the price of learning a substantial amount of someone else's code, the class library that you are using. As far as code reuse, studies have shown that in practice it doesn't really occur.

But there are other benefits to OOP. It allows the building of frameworks, which can provide a great deal of built in functionality with a minimum price to the user of the framework. This allows you to build on existing, tested code.

OOP also makes it much easier to write generalized code, that works for a wide range of data types (objects).

It's very useful having default behaviors that can be overridden when necessary.

OOP is a controlled, disciplined method to let similar situations take advantage of whatever common code they can.

You may not have thought of object-oriented programming as a "mere" human interface issue. But what else does OOP do but help the developer do his job?

There is some controversy about which is the best technique to do code sharing. What is clear is that we have a lot yet to learn in this area. It will take a few years for us to figure out the best way to do this.

5. Structured programming

Although this is not a new technique, it does belong under the "programmer interface" section. It's also worth mentioning since there are a few programmers who still don't use it.

6. Version control

Keeping track of revisions within the development environment is still a relatively new and rare idea. It's very hard to do well elsewhere.

[4]CASE stands for Compute Aided Software Engineering.
[5]Object-oriented programming is too involved to be explained here. I am assuming that most readers are familiar with the term.

Having ready access to changes that were made is very helpful in understanding where problems came from and in verifying that they are really fixed. They are also useful to have on those occasions when you need to recreate an older version of the software.

This would work very well with the concept of retaining testing information within the development environment mentioned in the next section.

"Source code" management also falls under this category and is important in keeping multi-

programmer projects from getting fouled up. I put the phrase "source code" in quotes because you still need this even if parts of your system aren't represented as code.

7. Data flow programming

This is another non-traditional way of building and looking at code. It's a good way to develop some kinds of software and it may be useful as another way to look at existing code.

8. Other aids to the programmer

a. Identifier location

Having the development environment able to immediately find who references who and to use this to navigate through your code is a very useful feature.

b. Aids to text input

If you've ever made a typo on a long variable or routine name, you know that it would help a great deal to be able to look up or navigate to get the right name. Some of us don't even spell that well!

c. On-line documentation

In addition to which routine to call with what arguments, you need to know something about what it does. You may also need documentation to figure out what routine to call in the first place.

Summary of programmer interface issues

Today's development environments have overwhelming amounts of information associated with them, especially as projects get large. We need all the help we can get in locating, understanding and sometimes hiding the details of today's systems. The right place to do this is within the development environment itself.

C. Testing, proving and debugging

1. Integrated testing

Debugging is a terrible name for what we actually do. We try to test code to either prove that it works or to find where it doesn't. Sometimes finding what actually caused an error is difficult and time consuming. It may not even be in our code.

We should be using debugging mainly in testing our software to see that it works correctly.

Our development environment could be a big help in this process. There's no reason why it can't keep track of what’s been tested, how it was tested and when. You also need to know whether anything has been changed since it was tested. Who knows, it may even be a good idea to specify the tests and the expected results along with the code in the development environment.

2. Super debugging tools

Debugging tools vary widely in quality. For any particular problem they either seem to work well or are barely useful at all.

The debugger should be source level. This does not mean just being able to step through lines of source code. You need to have the same level of access that you do in the writing process, including changing code while you're in the middle of debugging.

You need to be able to do cross-referencing, code editing and even the ability to write small pieces of code to aid in debugging. You should be able to flag this code as being part of this particular debugging operation and retain it in case you have a similar problem in the future. Traditional tools let you attempt this with conditional compilation. But it tends to make a mess of your source and you can't do it "on the fly." You have to stop everything and go through the edit, compile, link and debug stages.

D. Emerging techniques

Sometimes the hardest part of programming is coming up with a good algorithm or way of solving your problem. In some cases “trainable” systems avoid the problem of having to design a clever way to solve the problem, which may not even be possible. We’re even starting a to some hope that the computer can be used to design the algorithm itself. Either way help been given to what’s sometimes a very knotty part of the

problem.

1. "Trainable" systems

There are several technologies for making systems "trainable." These include neural networks, genetic algorithms and other artificial intelligence methods. This type of software saves the programmer from having to find or invent an algorithm that works for

these problems. Instead, by repeated trials the program learns which responses work best.

While this does not help the programmer write programs, it does save him from having to do a part of the job that can be very time consuming.

2. Genetically developed algorithms

A limited amount of work has been done in applying genetic algorithms to actually "evolve" some kinds of algorithms. The results have been surprisingly good and also very different from programmer designed algorithms. The biggest difference is that these algorithms are much less fragile than the ones people design.

This field has huge potential and some interesting ramifications. Can our egos stand it if machines can come up with better solutions than we can? There is the potential for programming to be done by specifying the results we want and letting the machine give us the best solution it can find.[6]

3. Beyond

Computers are amazingly flexible machines. The main limitation we face in finding better tools is not due to the computer. It's literally the limits of our imagination. This sounds like a phrase from a science fiction book but let's look at the problem.

With some very, very rare exceptions, all of our ideas about how to work with computers come from our previous experience with existing tools. All of our conceptions about what can, cannot and should be done come from this experience. We occasionally manage to break off on a short tangent when we are faced with an unusual problem. But for the most part we have a very hard time picturing concepts that are significantly outside of our experience. Once we get past our present set of limitations we'll find new ones. These will require new and different types of solutions.

III. What needs to be done?

There are limited things most of us can do to help solve this problem.

We must recognize the problem and the magnitude of its effects.

Programmers need to make management aware that this is a pivotal technology that multiplies the cost of all software development and limits our ability to develop, ship and sell products.

We should recognize that this IS NOT someone else’s problem.

If you do software development, you currently waste most of you time waiting for or fighting your tools. Life will be much more fun with better tools.

If you are in management there are serious wastes of resources and missed business opportunities due to this problem. Better tools will mean more opportunities that can be pursued with the same limited resources.

Let's see what we can figure out. We all have ideas about how to make our job easier. Let's share them and look for tools that will actually let us improve them.

A proposal

A platform is needed so that better development environments can evolve. Currently, whenever someone wants a better development environment, a whole new development world is created.

What should it be like? It should be independent of the programming language used and even able to express the same program in different languages. It should also support non-language views of code, such as graphical representations. It should definitely support object-oriented programming.

It also must be as open as possible to adding new development technologies later. We can’t foresee future development technologies but we can help make them easy to implement. We don’t want to lock out future advances if we can possibly help it.

[6]See Artificial Life by Steve Levy (Pantheon Books, 1992), pp195-204.

It needs to have symbol and structure level information readily accessible. This would make it much easier to do CASE tools, prototyping, visual/graphical programming, browsers, & who knows what else.

IV. Improved tools: Here now and on the horizon

Currently shipping tools

MCL (Macintosh Common Lisp)

Pluses

It's been shipping for several years.

Users report getting extremely good support.

It uses CLOS[7] to support object-oriented programming. CLOS is a very powerful and flexible when compared to C++.

Minuses

MCL is somewhat slow compared to C or C++ code.

It looks hard to learn to most programmers. (The things that scare most programmers, the language syntax and the parentheses, are not big problems.)

Common Lisp is a BIG language. There is a lot to learn if you want to know it well. (However, you need to know only a small subset to do most programming. Most Lisp programmers don't seem to know every last detail about it.)

It does not include an application framework. However there are several tools that make doing many standard Mac things very easy.

Minimum application size is large. Currently you have to take the whole development environment with you when you build a stand alone application.[8]

Improvements to MCL come out slowly. MCL should be considered a very stable and mature development system.

The Common Lisp code should work cross platform fine but Mac specifics won't. It does not support cross-platform applications.

Component Workshop

Pluses

It's been shipping since late last year, 1992.

It includes their own application framework.

It was designed from the ground up to be crossplatform.

Both the product and the framework seem to be improving fairly rapidly.

The minimum application size is around 300k.

It does not need make or header files.

Minuses

It is open to tools being built on it but with the same king of restrictions you would find with MCL or Smalltalk.

When you are ready to build a shippable application you must use the "Extruder" to produce standard MPW C code. This process is slow and potentially introduces problems that are new to the code you developed inside their environment. This is not the ideal way of doing things.

Component Workshop uses a custom (non-standard) subset of C++. They are working toward a more complete implementation but may be partially restricted because of the improved feature they support.

Component Workshop is currently limited to using their own framework. While they talk about possibly working with outside frameworks such as Bedrock, only time will tell whether and to what extent this is possible.

Comments:

While their non-standard use of C++ has some restrictions they also simplify some of the messier parts of C++. For example they have made multiple inheritance much easier and coherent to work with.

Symantec C++

This product was just announced at Apple’s World Wide Developers Conference. Since that was just before my final deadline I have not been able to take a very close look at it.

[7]CLOS-Common Lisp Object System

[8]They are working on this problem but it's a very difficult on to solve.

Assuming it's like other Think environments, it will have a better interface and much better turnaround than MPW C++. What we can expect is good but not astounding turnaround times and a debugger with a good interface and moderate capabilities. I don't expect any major new ideas or it to have easy ways for it to be extended.

Expected future tools

None of these products are yet shipping and the information on them is mostly preliminary, and in some cases is only based on rumors and logical assumptions.

"Pink"-Taligent's operating system

Taligent is the Apple/IBM joint venture that was formed to finish "Pink," the object-oriented operating system that Apple started.

Taligent is not talking much publicly about the details. From what little they have said, this may represent a major step in the right direction. They are focusing very heavily on being object-oriented. But if you listen carefully, they seem to recognize that there is more to the problem than just that. I just hope they are not blind by their faith in OOP or that the size of the project dooms it to non-creative solutions.

Stay tuned for more. Let's hope it's as great as they want us to think it will be.

Dylan

Dylan is a product, well maybe concept is a better word, of Apple's east coast Advanced Technology Group. They are shipping a manual (no official software yet) for what they are hoping will become a new standard development language. Initially, it had a Lisp-like syntax (actually Scheme, a Lisp style language). Non-Lisp programmers were apparently totally turned off and they are now working on a more traditional syntax for it. A new version of the manual will be available when this is done.

The intention seems to be to come up with a dynamic environment that is still efficient enough to use for a wide variety of tasks. Apparently, one of its major uses is for the Newton, Apple's announced, but not yet shipping, "Personal Digital Assistant."

Although details on the language are available, there are too few details on the development environment to make many intelligent comments. However, it's good to see Apple working on dynamic development tools. Personally, I think there may have been advantages to the flexibility of the Lisp-style syntax and I hope it is still available to those who want to use it.

MPW++ [9]

(The MPW replacement)

Apple has discussed its plans for a new development system to replace MPW. No schedule has been announced. The system is planned to be a modular, interactive dynamic. Tradeoffs include probably losing some of the spriptability of MPW. In its place we can expect a much better interface. Some new compiler technologies are being evaluated, but who knows what this means.

The bottom line is that this is mainly a big, but interesting, unknown. For it to make sense, I expect we'll see something in around a year. Openness to third party tools is an important feature. Depending on how well they implement it, this may be its most significant feature.

SK8

SK8 (pronounced skate) was shown at MADACON earlier this year. It was presented as being an authoring tool and was very impressive.

While it was not presented as a general purpose programming tool, it is an excellent example of what other tools can be like. We don't yet know to what extent it may be useful in general purpose programming.

In any case Apple has given no clues as to when it will be available, which probably means that it is at least a year away.

V. Well, will they be good enough?

Maybe I should try to answer the question I posed in the title. In one sense, I don't have enough information. Most of the products in

[9]This is a name I made up to make it easier to talk about. As far as I know no one else calls it this.

question are still vaporware (or in even just rumors and assumptions). At this stage, you're doing well to get any answers from vendors and what you do get are mainly stories about how great the tool will be when it finally ships. Until the restrictions and limitations of these products are known, we can only make good guesses about what they will really be like.

If I sound as if I'm avoiding the question, I'm sorry. There are still some answers. In fact there are two:

"probably yes" and "definitely no." I'm not being a smart aleck. These are two legitimate sides of the question.

"Probably yes":

With the number of new tools in the wings there will certainly be at least some significant improvement in the quality of tools available. Component Workshop, for example, does make major improvements in the C++, object-oriented development environment. The turnaround is much faster and it has a number of tools to help you navigate through the code.

"Definitely no":

On the other hand, the process of refining development tools will never be done. Assume we suddenly had all the features that we know about in our development tools. It would seem as if the fog had been cleared away. We would soon start noticing layers of problems that we'd never been able to notice before. Hopefully, our new tools would let us easily start building even newer tools to deal with the problems that had been obscured before.

Summary:

For our industry, this is a very serious and under-recognized problem. It makes our projects more expensive, take more time than they should and keeps us from accomplishing as much as we should. Fortunately, we are finally starting to recognize the problem and do something about it.

We still must recognize that we don’t know all the answers. We need to keep our development environments as open as possible so that we can incorporate as many upcoming new concepts as possible into them.

The tools I've seen and heard about improve turnaround but still have delays that can sometimes get up to from tens of seconds to a minute or so. This is an improvement but not nearly as good as can be done.

The other part of the problem that begs for more attention is the need to reduce the apparent complexity to the programmer. We can definitely use more tools that assist us in dealing with specific details as we are doing development.