Introduction to Thoughtful Programming and the Forth Philosophy

Michael Misamore

To the reader: constructive criticism and commentary is welcome!

Introduction to Thoughtful Programming

What is "good programming practice"? Why do most modern developers look to object orientation, polymorphism, scripting technologies, ultra-high level languages, component architectures, and libraries for "good" programming solutions? Are they good in the sense of end user speed? Certainly not. The modern programmer is more likely to sacrifice significant amounts of runtime processing power for a shorter development cycle. Are they good in the sense of complexity? Also obviously not; these new technologies are among the most complex solutions to seemingly simple problems ever created. Is abstraction a good thing? Computer science teachers would certainly say yes: they help the programmer get his or her work done faster by effectively providing snap-together components and a layer of glue called the "programming language". So some major trends in modern software development come to the forefront almost immediately: the tendency to sacrifice end user speed and complexity for ease and speed of development, and the increasing tendency to use abstraction to avoid the details of actual problem solving. In fact it seems that all one needs is a "general idea" of how to start solving a problem before one sits down to code. This is certainly the view of most computer science students.

This is also the view of businesses in the increasingly information-driven economy of the United States. They believe that software should be developed and shipped as quickly as possible to make it to the market on time. If this involves cutting corners in terms of runtime speed, code complexity, size, system requirements, or bug testing, it is considered a sacrifice which can be remedied after the product ships. After all, the business has a bottom line, and if the code works, marketing is perfectly willing to sell it to people.

Now, it is doubtful that many programmers, perhaps after some hard thought, would consider this kind of software beautiful. It is certainly suboptimal in many aspects already mentioned, there are certainly sections of code which could be rewritten or done away with, perhaps it was written in a scripting language when it should have been written in a lower level language, etc. And let's not forget the hundreds or thousands of bugs that still lie in the code because the deadline was hit. This is not the kind of code that a programmer would be proud to admit to writing.

Perhaps there are some things to be proud of in the technologies used in the product itself. "I designed a new class hierarchy which makes much more sense than the previous one" or "our new reusable component architecture makes communicating over the net a snap!" or other similar responses are probably common. But are these architectures actually necessary? What has the programmer actually accomplished besides creating new layers of complexity atop the previous layers? Certainly it is

possible to accomplish these tasks without abstract architectures and garbage collectors and network layers and applications programming interfaces and maybe even operating systems? What victory, then, may be claimed here? A victory over complexity? No, since this new technology just adds complexity to the machine. More likely this was a victory for ease of programming. This was a victory for the programmer rather than the end user. This was a case of the programmer being a little greedy, wasting the time of thousands or millions so that he can waste a little less time of his own.

So in fact there is very little to be aesthetically proud of in the development of commercial software as it is currently developed, besides the fact that it gets the job done, or, at least gets the job done sometimes, in some circumstances. Programming aesthetics take a back seat to the industry's demands, and this has usually resulted in software of poorer quality than is necessary. The philosophy taught to the mainstream of computer science students emphasizes these industry goals, and asserts that these are good and "right" for use in software engineering. However, there is another school of thought, one which has been around as long as computer hackers and enthusiasts have been around, which emphasizes a very different style of programming. If taken seriously, this older set of aesthetics leads to an entirely different philosophy: the philosophy of Thoughtful Programming. More on this later. First lets discuss the problems of the more modern industry philosophy.

For years now, the computer industry, largely in marketing efforts, has given the average consumer the impression that computing technology is abundant and ubiquitous [4]. Neither is really the case. Comparatively very few people in the world even own a personal computer and those who do mostly live in only the richest countries in the world. To the average person an Apple IIe or Commodore is a dream machine. This trend in marketing tends to encourage wastefulness both in the industry and at the personal level. In the latter case, good hardware and software is simply abandoned or destroyed. Computers clocked at 75 megahertz are considered so slow as to be unusable (this is due to increasingly slow software). This is of course the end result of the tendency for the industry to produce slower software (which wastes more clock cycles by definition) since the hardware is also improving at break-neck speed and, of course, old computers are considered to be disposable.

Indeed, a number of new technologies have arisen to fill all of these new clock cycles, and more often than not they benefit the programmers and executives more than the public in general. These new bundling trends and technologies include object orientation and rapid application development products, both of which have no positive impact on the end user (besides, perhaps, less bugs). One only needs to look to computer boot times to see how software has actually regressed in some sense over the past twenty years [4]. Observe the old Apple computers booting in only a few seconds versus a modern Pentium class machine taking almost a minute just to bring up the simulated graphical environment. One might object at this point that the latter environment is more sophisticated and hence will naturally take more time to load, but exactly what is gained from this increased sophistication is hard to tell.

It is also important to note that these new technologies which waste the resources of modern machines are also sources of lots of new problems and bugs. As every programmer probably knows, the complexity of the problems one has to deal with often increases exponentially with the size of the program being written [7]. Even in a component based architecture, unexpected interactions between the underlying components and misdocumentation of functionality lead to many headaches. Add to this the problem that the increasing complexity of computer hardware and software increases the job

security of the average programmer, and hence gives the programmer an incentive to raise the amount of abstraction and abstruse methodologies to stratospheric heights [6].

Due to current programming methodologies, the average programmer has certain expectations for the types of programming tasks put to him or her. Often a programmer may be asked to effectively "glue" existing components together to implement a new application using largely already-existing code. This leads to more or less drone-like programming since writing code to combine existing pieces of code together into a larger whole usually presents no special challenges or difficulties. This leads one to wonder whether this is a result of the redundancies built into the workforce. As Chuck Moore has put it: "You can't replace the smart people, but you can replace the dumb people" [7].

The language that the programmer uses usually has a syntax which is designed to prevent the programmer from making mistakes (or otherwise the syntax seems to be thrown in more or less arbitrarily for no reason other than to stay consistent with a traditional language like C or Java). Thus on both these accounts the programmer is encouraged to think as little as possible in the actual process of programming and the task resembles more of a "gluing together" experience rather than a "building from scratch" experience.

Going back to complexity and job security, the average programmer expects whichever language is employed to take months or years to learn and master. This expectation is also largely a result of the bundling technologies which were originally supposed to simplify programming. Techniques such as late-binding are used to correct problems in these existing methodologies, adding yet another layer of complexity and decreasing actual runtime performance [9]. The average programmer expects lack of communication with the hardware designer, because the predominant viewpoint is that software and hardware should be orthogonal. This is essentially the "write once, run anywhere" philosophy. Compilers are expected to write machine code for the target architecture, and almost invariably do a worse job than a good human programmer could do (the only exception being when the hardware is so phenomenally complicated that no single human could possibly take into consideration every possible optimization). Finally, the average programmer expects a huge set of "standard" libraries which simplify the task of programming, at least to the programmer. Modern high level languages, despite various sets of language features, resemble little more than glorified "wrappers" for these existing libraries [9].

All of these phenomena may be traced back to a central theme: Chuck Moore has referred to it as the User Illusion. This is the illusion that computers themselves are actually the abstractions which they implement [4]. It is the idea that computers consist of desktops, windows, files and their filesystems, operating systems, and so on. This is the paradigm Apple has taken with their Macintosh and later Microsoft with their Windows operating system. The goal of such an effort appears to be to mask the increasingly complex underlying technology with a layer of abstraction which is intended to give the user the impression that the machine is less complex than it actually is. This is believed to promote a more "user friendly" experience, where the user has an easier time operating the machine. However, as the massive amounts of technical support and user frustration seem to suggest, this may not be the case. Often support problems arise because the user has no understanding at all of how the hardware actually works and what it is designed to do, or because the complex layers of software on the machine are too hard for the lay user to understand the underlying problem. Hence it might just as easily be argued that the increasing complexity of hardware and software have resulted in more

difficulty in operations of a personal computer, not less [4].

The User Illusion also has implications for how the computer is programmed, as mentioned above. Tendencies in the design of modern software may lead the programmer to believe that the only way to accomplish the task at hand requires some complex methodology, perhaps because the programmer is attempting to anticipate problems in the general design of the software. Sometimes it is the case that the software has become so complex that these problems are in fact internal to the software design itself, and have nothing at all to do with the actual problem to be solved. These are referred to by Thoughtful Programming practitioners as the "non-problems" which are to be avoided in the design [3]. The industry programmer finds these "non-problems" to be barriers to problem solving because of the orthodox methodologies he or she attempts to employ.

Another set of problems has arisen because of the non-thoughtful practice of believing that the program that a programmer is working on is going to be the only program running on the machine of the end-user. If the average programmer pauses to think about it, one probably focuses all of one's attention on the aspects of the program's own runtime behaviors, and completely foregoes, even in the design of the program, the concept that perhaps this will not be the only process running on the machine. If this is the paradigm that every programmer follows in the design and implementation of new software, it is no wonder that programs are not designed to have minimal impact on system resources; after all, the programmer implicitly assumes that all system resources belong to him or her. All of this refers to the more common preemptive style multitasking in modern operating systems. Perhaps in a cooperative multitasking system the User Illusion has a less negative impact along this axis.

The User Illusion is also somewhat responsible for the belief that hardware and software must be very complex in order to accomplish useful tasks. History shows us that this is not the case, and indeed most of the extra complexity arises from the aforementioned industry philosophy of patching continually instead of redesigning existing hardware and code [1]. Marketing to consumers has convinced the average customer that faster always equals better; this is largely the result of software becoming progressively slower, leading to a vicious circle.

It is also (very mistakenly) believed by many that abstractions used in programming must have some actual presence in the object code which is produced. Tables of virtual methods from C++ and late binding/polymorphism again come to mind as manifestations of this development philosophy. Instead of developing the software in a way which conveniently processes abstractions at edit time in a metaprogramming framework, current software instead implements abstractions directly into the final object code, leading to slower runtime performance and larger memory footprints. This line of thinking tends to encourage pushing design time tasks to edit time, edit time tasks to compile time, and compile time tasks into runtime. That is, programmers tend to want to sit down and start coding immediately, making many mistakes which could have been avoided with a thorough design and redesign phase [3]. While programmers code, they don't tend to think when they are editing the code and hence wait for the compiler to spew out a list of errors at them, wasting additional time as the programmer switches out of the editor, gives the correct command for compilation, tries to interpret the error messages, and goes back to the editor again. It is important to note that throughout this time very little actual work is being accomplished. As mentioned before the final step is to use libraries and abstraction technologies to push what work should be done at compile time into the actual runtime of

the program itself. This process obviously tends to make the program more complex than is needed and results in programs with sub-par performance.

Now that considerable thought has been given to the problems in the larger school of programming philosophy, the paradigm of Thoughtful Programming arises as one possible alternative. The fundamental difference between the philosophy of the larger computer industry and that of Thoughtful Programming is that the latter school has as its goal "minimizing overall complexity" [7]. This has required efforts, on both the hardware and software fronts, by individuals such as Chuck Moore and Jeff Fox, among undoubtedly many others. Their main principles are to create faster, simpler hardware and software solutions than those which are currently available, and in fact to produce them at competitive prices. This requires a dramatic shift in the methodology used to develop new hardware and software. In the former paradigm, focus is placed on which features to put into the new design, whereas in Thoughtful Programming the emphasis is on what can possibly be removed to make the proposed solution simpler in the sense of overall complexity [4]. The latter methodology encourages development of entire solutions, not simply new software upon inadequate hardware. The hardware must be general enough to apply to a wide variety of possible applications, and yet simple enough that the average programmer can understand every important aspect of the machine's behavior. This type of deep understanding of the machine shatters the User Illusion, and is an approach which is nearly unheard of in mainstream software development for personal computers.

In trying to design a solution to a programming problem, the Thoughtful Programmer starts with "a clean sheet of paper", as Chuck Moore puts it, and starts to think about the simplest possible solution to the problem at hand [7]. A very important aspect of this process is to identify the "non-problems": those assumptions about the necessary existing software technologies or methodologies needed to produce a satisfactory solution. Due to the much more direct assault on the problem the Thoughtful Programmer takes, it will be unlikely that any of these technologies are optimal for the specific problem at hand. At this point, the Thoughtful Programmer becomes a creative artist, and attempts to produce a solution which avoids solving all the non-problems standing in the way [3]. Representing the proposed solution in the least complex manner possible then becomes the priority, and for this the Thoughtful Programmer will write code, test it, try to break it, rewrite it to make it simpler and more elegant, and test it again. This cycle will repeat until the Thoughtful Programmer is aesthetically satisfied with the result [3]. In order to facilitate this process, it is useful to have a programming language which enables the programmer to design and test new ideas as rapidly as possible, ideally in an environment which allows new functionality to be tested as soon as it is written. Also, it is important for the programming language to be extensible so that it may easily be molded to suit the problem at hand. And by all means the language must be as flexible as possible; the programmer cannot be hindered by any inherent limitations in the expressive capability of the language itself.

Now lets observe how a Thoughtful Programmer goes through the stages of Designing, Editing, Compiling, and Running new computing solutions. Design of the solution should take the most time to complete of any of the steps, and it is the one which must be repeated most often. Each new solution should be critically examined for possible algorithmic or structural improvements, and these improvements must be integrated and tested thoroughly to make sure that the solution is indeed optimal. Remember that the biggest algorithmic improvements come not from employing the obvious and straightforward industry methods, but instead from creatively producing solutions which uniquely fit the problem at hand. Also remember that the truly optimal solution may come through the design

or use of new or different hardware. Do not be afraid to consider radically different technologies as possible options for problem solving [3].

Once a satisfying solution has been arrived at, the editing phase begins, and this is the first step in the actual coding of the software. Ideally the development software will allow for the design and testing of as many ideas as possible in a short time frame. It is also at this stage that the Thoughtful Programmer must concentrate efforts on eliminating compile time and runtime overhead through the philosophy of Early Binding. From a conventional industry standpoint, Early Binding is a bad idea because it makes the final software components more inflexible and wastes compile time because modifications need to be made to each object which requires change. These objections are moot to the Thoughtful Programmer for two reasons: first, the Thoughtful Programmer does not worry about flexibility at this stage because any such changes will usually require sacrifices in complexity and runtime performance; and second, the software that the Thoughtful Programmer tends to write is of such minimal complexity that it is easy to identify and fix any code which needs to be changed. After all, the Thoughtful Programmer is generally NOT interested in software reuse; the more immediate and important concern is that the software solution being developed right now is optimal. New and better editing technologies may help alleviate any perceived problems with Early Binding as well; Early Binding need not make actual code more difficult to read or write. Various metaframeworks may be possible which might bind code early and eliminate redundancies, all while preserving the programmer's desired level of abstraction and control over the final code produced. More work should be done in this direction.

To the Thoughtful Programmer, editing is about representing the desired solution in as simple and effective manner as possible. Hence a similar process to the Design stage should be implemented, where the fat is trimmed from the representation as much as possible [1]. This methodology also stresses the importance of willingness to redesign: if the code being edited is suboptimal as a result in a flaw in the design, the programmer should be willing to start from scratch with a new design to fix the problem. This should not be that big of a problem if the Design has been well thought out in the first place, since the programmer is already well familiar with the problem and should be aware of how to recode the small amount of code necessary to implement the new design (and this code will almost always be relatively small if an efficient representation is used). The final coded solution should be as "brutally simple" as possible, and should try to do as much work in representation as possible so that less work needs to be done at compile and runtime [1].

If the Thoughtful Programmer has done his or her job right, Compile time should be a snap. The key is to code the solution so as to compile as little as possible and to interpret as much as possible before the user actually has to run the code. If the language is simple, the compilation will be fast, and this is obviously a desirable feature to have. Numerical constants should be computed at this time, not at runtime. If there are bugs, the language should allow the programmer to quickly isolate and fix them. If a bug arises which necessitates a change in the Design, it will be necessary to go back and redesign. Again, it is desirable to redesign: a good programmer will be proud to make the code better if possible. This is what distinguishes coding as an art from coding as an industry. If the steps have been correctly followed up to this point, the code will be a solution of minimal complexity necessary to solve the problem at hand. In all likelihood, runtime performance will probably be through the roof compared to competition.

Forth: A Language for Thoughtful Programming

The programming language Forth sticks out for representing many of the desired features of a language which would be useful for Thoughtful Programming. Forth was invented by Charles Moore in the early 1970s and has since become popular among Thoughtful Programmers for its emphasis on simple syntax, speed, ability to function at both low and high levels of abstraction, portability, extensibility, and general ease of use in development. One of the things that makes Forth exceptionally pleasant to newcomers is the fact that the language is very easy to learn [7]. In contrast with programming languages in the larger industry arena which take months or even years to learn adequately, learning most of how to use Forth only takes about a half an hour. The syntax is brutally simple, consisting of only words and whitespace, removing barriers that in other languages tend to hinder any remaining creative spirit programmers might have [1]. Hence the language as written has as its focus semantics rather than the peculiar syntactics of other languages [2]. This also makes the language particularly flexible and extensible since the syntactical rules do not restrict, and in fact promote, the employment of entirely new languages and rules within the existing system to express solutions.

As a result of the simplicity of the language itself, Forth is extremely easy to compile compared to other languages, and it is easier for the programmer to express exactly what he or she wants the machine to do. The language is ideal for programming in unison with a given machine's hardware, yet it has such a simple implementation on the machine itself that it can really be thought of as a high level version of assembly language [9]. Forth is very capable of low level programming tasks in particular because of its complete lack of restrictions on fundamental data types; in fact, there are none besides the standard "cells" and "double cells" which are basically units of raw memory. Hence the programmer is free to implement his or her own data types as freely as the machine may allow, in a syntax which is more powerful and convenient than assembly, and yet just as capable of direct communications with the hardware. Forth is so hardware friendly, in fact, that the primitives of the language itself may be implemented in micro-chips which are 1000 times less complex (in terms of transistor counts) than competitors, bridging the gap in simplicity between software and hardware. This is the Thoughtful Programmer's dream: to have hardware simple enough that it can be completely understood by a single human being, so that any software written for it can truly result in computing solutions of minimal complexity.

The fact that the primitives of the Forth language may be implemented in hardware does not mean, however, that it is not an effective high level language as well. In fact, the flexibility of the language allows a Thoughtful Programmer to implement whatever layers of abstraction are necessary to solve the problem at hand [9]. Since the possible data types are unlimited, the capabilities to effectively handle problems in packet mangling, databases, filesystems, and so on are effectively infinite, and the extensibility of the syntax of the language itself allows for the most convenient representations of these structures that are possible.

A Forth system as implemented in software is effectively a virtual machine. It has all the power of the Java virtual machine and more, thanks to its complete freedom of data types and lack of additional structure like objects and arrays which are provided even if the Java programmer does not really need them. Unlike Java, the Thoughtful Programmer has complete freedom in how to implement a Forth solution on the target platform, and has an inherent ability which comes from the foundations of the

language itself to interact directly with the hardware. The Java virtual machine may be based on similar technology to Forth virtual machines, but the Java philosophy of implementing complex solutions on a virtual machine so that the programmer does not have to reimplement the solution for different platforms is completely against the style of Thoughtful Programming.

This is not the only way in which Forth dramatically differs from Java and other modern programming languages. Forth has traditionally taken the role of the user/programmer operating environment of the machine being developed upon, and hence has built into it a notion of direct programmer interaction in the software design process. The Thoughtful Programmer demands that software ideas be able to be written and tested quickly, and that new functionality may be tested immediately for its utility in a given programming situation. No other language could be better suited for this purpose. Forth is completely interactive with the programmer, giving direct access to the compiler and interpreter, including the compiler's own behavior. The Thoughtful Programmer has to freedom to tell a Forth system when to begin compilation, how to proceed with it, and when to pause or end it. As soon as new "words" are created, their functionality may be tested immediately without switching into a different environment. Effects on arbitrary inputs may be checked as often as desired without leaving the programming environment, making debugging a snap.

The freedom of Forth's syntax is sometimes criticized as allowing for code which is harder to read or maintain than in more traditional languages [9]. This criticism may be argued against for a number of reasons. First of all, as mentioned above, one of the advantages of Forth in software design is the inherent flexibility in the syntax. Such flexibility tends to lead to much simpler solutions to software problems due to the extensible nature of the language itself, and, if done right, might actually result in programs which are easier to read than their C++ and Java counterparts. This results from the unique ability of Forth to describe in words what it is doing as it does it. As for maintenance, Forth programmers tend to be Thoughtful Programmers and hence tend to write solutions which are as simple as possible using representations which are as efficient as possible. If a change needs to be made, it therefore should not be that arduous of a task, and if the change is large then the programmer should think about rewriting the system anyway.

In addition, Forth, like any other programming language, may have the misfortune of having particularly bad programmers write and promote bad coding solutions. It is easy to write equally horrendous code in just about any language that one might choose; Forth does not stand alone in this department [9]. The moral to take from this is that one bad coding solution in Forth is one bad coding solution in Forth; such examples rarely serve as good indicators of the readability of the language, which itself is a highly subjective property anyway. To help combat readability problems, good Forth programmers employ "factorization": the practice of breaking a program into very small bits of code which may be coded and tested individually [7]. The idea is that new word definitions can be one or two lines long, which is much more readable than, say, a sequence of 50 consecutive words. Forth uses a stack to pass inputs and outputs between words (remember that data structure from computer science?), so all arguments are implicitly passed, making the language both easier to read and faster at runtime (than say, languages like C which must set up new stack frames for each function). And of course, there is always a need for good documentation for software, and Forth systems are no different. Each word should be individually documented for its effects on inputs and outputs, and it is good practice to keep the words simple enough that stack diagrams (comments on the structure of the stack as different words act) are not needed. Remember that simplicity in representation is one of the

core aspects of the Thoughtful Programming paradigm.

Another reason that Forth's simple syntax is an advantage to the Thoughtful Programmer is that doing so pushes the complexity from compile and runtime back to edit time. This is exactly the type of "pushing work backwards in time" which is advocated by Thoughtful Programmers [7]. The point is that the language itself need not be complicated to accomplish useful work. Often times the editing environment knows a lot more about the overall structure of the project, and hence it makes sense that as much important preprocessing as possible be done at the time the code is edited.

One reported problem that many programmers may initially have with the Forth language is its used of reverse polish notation for arithmetic [9]. This means that the operators on arithmetic expressions appear after the operands, not between or before them, and it is the mode commonly used to evaluate arithmetic expressions on HP calculators. This type of computation is inherent to Forth because of the central role the stack plays in the language. Such expressions may appear confusing at first, but like many technologies employed for Thoughtful Programming, the purpose is to simplify operations and dispel the User Illusion. Simplification of arithmetic expressions in RPL occurs because of the lack of need to parenthesize expressions as they are written. Expressions which might normally be mathematically expressed in terms of many parentheses are instead reduced to the natural order in which a computer would actually evaluate the expression, and thus gives the programmer an upper hand by having full control over the exact order of computation. This can be especially handy when managing things like significant digits. And even if the curious industry programmer cannot get the hang of RPL notation, the Forth language provides the flexibility to implement his or her own personal arithmetic syntax, which could be infix, prefix, or any more elaborate scheme.

There are doubtless many other objections from programmers who are skeptical of the Thoughtful Programming paradigm and the Forth programming language. It would be informative to the casual reader to discuss what they are here. One popular and quite general objection to these methods is that layers of abstraction are necessary because the hardware which one must implement software for is too complicated [4]. The way a good Thoughtful Programmer would probably react to this objection is to simply state that the hardware should be made simpler. After all, if the hardware is made less complicated, the programmer will have an easier time programming directly to the machine, there will not be a need to worry about pipelines, caching, and out of order execution. All said technologies are just extra layers of complexity which are added to the already tremendously complex CISC and RISC style microprocessors.

Despite all objections to the contrary, the Thoughtful Programmer will probably assert that even RISC chips are much too complicated to work with; after all, it is claimed even by good programmers that C compilers for RISC code will outdo even good RISC assembly programmers. This is not at all a desirable situation for the Thoughtful Programmer: the goal is to have a machine which is simple enough for a human to understand, and RISC is not simple enough for these purposes. Custom Minimal Instruction Set Computers known as MISC microprocessors have been designed by Forth and other stack computing enthusiasts as viable candidates for truly simple computing. One of the key ideas in such designs has been to implement Forth as the machine language of the processor, and to use Forth's radically extensible nature to implement any of the more complicated operations which are necessary. The processor is the high level language and the implementation is far easier to produce than that of more complicated languages like Java.

Another very common and general objection from people who may or may not understand how Thoughtful Programming works is "I would like to use method X, but Forth makes it very difficult to implement" [4]. This type of objection would probably come from someone who has been trained in the object oriented or functional programming languages before coming to consider Forth. There are a couple of points for the objector to consider. First, one has to consider whether this method is actually necessary to implement the solution. Chances are it will not be absolutely necessary, and in fact may get in the programmer's way if the programmer continues to do it "the hard way". In Forth, implementations of more advanced methods are often more difficult because Forth is closely tied to the machine and hence reflects the actual complexity that the machine will be working with. This has the positive effect of encouraging the programmer to simplify his or her solutions so that an actual machine could be efficient in implementing them.

The second point is, if the programmer is interested in using, say, the object oriented approach, then the programmer is free to implement it however he or she wants [9]. The utility of good factorization in writing code should not be underrated. Breaking the problem into smaller pieces which may be individually developed and tested allows for even complex solutions to be implemented with a minimal amount of fuss. Writing any project can seem overwhelming if the programmer does not take the time to write a design which breaks things down into parts. This is abstraction put to good use: not for code reuse or isolation, but for testing the implementation piece by piece as it is built.

Future Directions in Thoughtful Programming and Forth

Over the past 30 years, Forth has been evolving, largely due to the continuing work of its inventor, Charles Moore. While many programmers have attempted to standardize Forth as it was practiced 20 or 30 years ago to crystallize the language to outsiders, Chuck has continued to develop new and exciting ideas in Thoughtful Programming. His new colorForth language and development environment is a reflection of his ongoing search for a more convenient coding environment for the Thoughtful Programmer. An expert Forth programmer by the name of Jeff Fox has also made new and exciting contributions to the Thoughtful Programming paradigm, especially through the design of his new aha Forth system. Some of the ideas that these individuals have been working on are quite revolutionary, and show that cutting edge Forth is smaller, cleaner, and faster than ever. Indeed, over time the language itself has had the tendency to become simpler!

One new idea being tried is to completely avoid the storage of object code, and instead to compile source on the fly. This provides an obvious portability advantage in Moore's colorForth. Such an advantage is made possible through the simplicity of the implementation of the language: compilation takes no perceptible time to complete [8]. This is a result of a recent key innovation in Forth designs: to store the source code in an unconventional format. Recently these innovators have realized that Forth source code is about words, not about characters, and hence that Forth source is not ideally expressed or handled in plain ASCII format [8]. Instead, alternative representations of source based on "tokens", roughly word references with additional information appended for editing or compilation purposes, are used. This preparsed source allows for dictionary searching to be eliminated from compile time, so the laborious process of searching for ASCII strings in a word dictionary is completely eliminated from the compilation process. Compilation speedups due to this new technology have been dramatic [5].

Another recent innovation in Forth systems is due to Chuck Moore, who realized that in the simplification of source code, color could be used as a meaningful way of communication information about the behavior of the various words being displayed. By placing special information in the whitespace of the source code, Mr. Moore has been able to express his source in a simpler and easier to read format. The various colors have also been used to give words individual behaviors, eliminating much ambiguity in the behavior of how the source listings would function when compiled [8]. While Moore's colorForth was not initially designed for a wide audience, he has recently made some efforts to make it run on a wider range of personal computers than just his own systems. The outstanding efforts of Sean Pringle to bring colorForth to the masses through the Enth and Flux projects will provide a ripe test-bed for new ideas. Developments along these lines have tended to stress simplicity, avoiding the massive operating systems and BIOS systems in modern personal computers as much as possible [5].

As Forth source code has taken on completely new representations in the recent systems, more sophisticated editors have become necessary. User interfaces for these new editors are still highly experimental and under constant development. The general consensus among Thoughtful Programming enthusiasts has been to push things backwards in time in the traditional Thoughtful Programming style, performing source code compression and optimization at edit time rather than at compile time [5]. As these systems continue to develop, one might expect to see better error detection at edit time, debates about when and how to pack the source code, new optimization techniques, and a variety of different scripting technologies or different means of representing source code. Whatever the future holds, Thoughtful Programmers, likely using Forth and its descendents, will be finding new and innovative methods for minimizing complexity in computing solutions for years to come.

References

[1] Fox, Jeff. "Low Fat Computing". December 6, 1998. Online, Internet. Available http://www.ultratechnology.com/lowfat.htm

[2] Fox, Jeff. "Forth - the LEGO of Programming Languages". November 24, 2000. Online, Internet. Available http://www.ultratechnology.com/4thlego.htm

[3] Fox, Jeff. "The Forth Methodology of Charles Moore by Jeff Fox". December 9, 2001. Online, Internet. Available http://www.ultratechnology.com/method.htm

[4] Fox, Jeff. "Thoughtful Programming and Forth". Online, Internet. Available http://www.ultratechnology.com/forth.htm

[5] Fox, Jeff and Sean Pringle. "ENTH Flux aha Color Forth". Online, Internet. Available http://www.ultratechnology.com/enthflux.htm

[6] Moore, Charles. "1x Forth". April 13, 1999. Transcript courtesy of Jeff Fox. Online, Internet.

Available http://www.ultratechnology.com/1xforth.htm

[7] Moore, Charles. "Moore Forth -- Chuck Moore's Comments on Forth". January 25, 2001. Transcript courtesy of Jeff Fox. Online, Internet. Available http://www.ultratechnology.com/moore4th.htm

[8] Moore, Charles. "Philosophy". July 20, 2001. Online, Internet. Available http://www.colorforth.com/phil.htm

[9] Various Authors. Comments from the slashdot.org story "Chuck Moore Holds Forth". September 14, 2001. Online, Internet. Available http://slashdot.org/article.pl?sid=01/09/11/139249