The Force Operating System and Software Design Project

A Software Design Paradigm Emphasizing Portability, Efficiency, Stability, Security, and Customizability through Modularity

by Jonathan Jaekle

Introduction

The current system controlling software development has reached its limits. Huge applications such as Microsoft Word, Adobe Photoshop, and Realplayer are unstable, memory-inefficient, and are not customizable. They are unstable because if any one of the millions of lines of codes is to cause a serious exception (protection fault), the entire program is terminated, often with loss of data. They are memory inefficient because tons of code for features that you will never use is being loaded into memory whenever the program is run. This is not to say features are not good things, but rather that obscure features should be located in separate programs, and could be loaded only when the user wished to access them.

These programs are not customizable due to the fact that they are designed to be the "end all", to satisfy all needs of the user. The main advantage to this system is that software companies are able to make more and more money off of new versions, without allowing third parties to create more useful, intuitive, or efficient features. This essentially prevents true competition in the software market. Software companies which make one popular piece of software are able to ride on it's coattails for years afterwards, because even if their newer versions are of poor quality, people will continue to buy them so as to keep "compatible." In fact, software companies try to make their software the only software that can work with their documents. They also attempt to hide completely how the program works, and usually prevent any third parties from developing plug-ins (except in certain extremely restricted areas, such as Photoshop filters.) This forces people to use their product, instead of going to a third party for a feature, which may develop it sooner, more efficiently, or at less cost than the large company.

The Microsoft Windows family of operating systems is an example of this disturbing and greedy trend taken to horrifying extreme. While windows does possess some degree of what one would call "modularity", in that there are hundreds of Dynamic Link Libraries, this modularity is not taken advantage of. A program calling a DLL will still "crash" if there is a protection fault in the DLL. This is unacceptable. Users rarely have the option of deciding which DLLs are loaded into memory, and so there is a great amount of wasted memory. Windows itself contains many large, buggy applications, such as Internet Explorer, Windows Explorer, and Windows Media Player, all which crash quite often.

This monopolistic strategy is great for software companies, but horrible for users. They end up spending more money for software that crashes more often, uses up greater amounts of memory, and can do less than a system designed around modularity. The modularity system attempts to break up software into application independent modules, which exist as separate programs in memory. They communicate with one another only in well-defined and error-checked ways to perform the desired tasks, and can be arranged in whatever order and configuration the user desires.**The Force** is an operating system and software design paradigm with the goal of being portable, efficient, stable, secure, and highly customizable. These goals can all be achieved through the extensive use of modularity.

Modularity

As it stands now, programs are generally written as a collection of "functions," which perform a task. The problem is that as programs get more and more complex, and hence composed of more and more code, the chance for bugs increases. When all the functions exist in one executable program, and all share access to the same address space, an error anywhere can bring the entire program down. Modularity essentially breaks down a program into much smaller collections of functions, which are each actually separate programs, and these programs/modules have a well defined task, which they perform to the exclusion of all other tasks. When an error occurs, it occurs in only one module, and an error code is returned to the other modules, which can take the appropriate steps to continue running, without terminating themselves. Loss of data would occur exponentially less often, due to the fact that there would only be one module that would actually be holding the e.g. currently edited document. This module would presumably be quite small, and quite carefully written. If any module other than this were to have a terminal error, the data would be left unaffected. Also, modules in The Force will have well documented and openly published interface specifications, so that anyone will be able to write their own module to replace an expensive, inefficient, limited, or otherwise unsatisfactory one.

Efficiency

Efficiency in this context does not refer to the speed with which the program is executed, as modularity, at least at first glance, appears to be slower in this respect (though there are ways to get around this that will be discussed later.) However, processor speeds are already at levels that would have been unheard of ten years ago, and there are no indications that this rapid escalation in performance will slow. Efficiency refers here to overall efficiency, in terms efficient use of the users time, and this is a far more valuable resource than simple processor efficiency. It also refers to efficient use of programmers time, which can be very costly indeed. With a modular system, creating a small program is drastically easier, as almost all tedious tasks (choosing files, setting up toolbars, setting up menus) are already taken care of modules. Thus there is much less duplication of effort, which is what plagues our software system today. Lack of duplication of effort also means that bugs only get written once, and then a bug fix is permanent, rather than having different companies write the exact same code, the only difference being different errors.

User efficiency has great potential for increasing under the modular system. Let's suppose we have a spellchecking "module", that can be used in absolutely any program that uses text input. It would obviously find utility in a word processor or presentation program. But also important would be it's use in an email program, and instant messaging program, and even web page input or file naming. Now, under the current system of software development, the large programs: namely word processors, presentation (e.g. PowerPoint) programs, and email programs have spellcheckers, and if they are from different companies, they are different spell checkers. Which means that the user has to learn two different spell checkers, and has to use one that is inferior. Of course, the instant messaging program, web input, and anything else that could benefit from spellchecking go without it. Why? Because the spellchecker from one of the spellchecked programs cannot work with anything besides that program (and that's a design goal of the software companies.) And also because, even if one were to write a spellchecking module, the instant messaging program and web browser do not allow one to at all easily add components to enhance functionality. Of course, a software company's response would be "Oh, we'll include it in the next version." So the user has to learn to use two different spellcheckers, and is left without spellchecking for every other program. Were the spellchecker a reusable module and the programs openly customizable, the user could select their favorite spellchecker and use it for all their applications. This all adds up to increased efficiency.

Stability

The increased stability of overall programs and hence data has already been discussed. But this applies equally, or perhaps even to a greater extent, to the operating system as a whole. Since only a few modules need to have direct access to hardware, namely drivers, and only a few modules need direct access to those, the actual hardware can be protected behind a few very secure modules. Every other module has no access to hardware or device drivers, and hence the chance of an actual system crash can be made to approach 0. Since modules are so small, advanced logic programs can be developed to logically prove that a module cannot crash, provided hardware works as documented. An even more striking example is that viruses could be virtually eliminated. Only a very few modules should have the ability to delete files (except their own), rename files, or write to executable programs . Therefore, any downloaded program would not be given these rights, and would not be able to infect other programs or delete files. That this extremely simple idea has not yet been implemented in the Windows family of operating Systems is quite disturbing indeed, considering antivirus software is a multimillion dollar industry, and millions of dollars in man hours and repair fees are lost yearly to viruses (makes you wonder if perhaps Microsoft and Norton have some sort of "partnership.") Stability itself directly contributes as well to efficiency, as great amounts of data and hence time and money can be lost due to system crashes, program crashes, and viruses.

Security

The same idea behind protection from viruses applies to protection from outside threats. Most hacking exploits rely on bugs in the system, and it should be apparent from the last section that any bug's effects should be greatly reduced when utilizing modularity. Each module is essentially a separate fortress, checking any incoming data for consistency, and only passing on this data if it is in the proper format. Even if a hacker were to penetrate the security of a web or ftp server, the program would generally not have access to any files but its own and the serving files, and often only read access to many of the (more important) files. The idea of limited access for each module also yields great security against users with limited access. Analogous to virus programs discussed in the last section, restricted users themselves would not have the ability to delete files that were not their own, and would certainly not have the ability to edit system executable programs. All interaction with files would be done through a file serving module, that would check their access rights, and prevent any unauthorized access, and all other potentially dangerous interactions would be done with analogous server type modules. All programs that were run by this user would inherit his restrictions in addition to their own, and this would likewise prevent illegal interactions. Increased security leads directly to increased efficiency for obvious reasons, and it also increases stability.

Customizability

Customizability (which generally refers to superficial things such as colors or sounds) is probably better thought of as "constructability": the ability to construct novel software solutions from pre-built components. One of the only truly useful upgrades to the Microsoft Word system in recent years has been the addition of an "autosave" feature, where the edited document is saved to a temporary file whenever there is idle time. When a crash occurs, the temporary file can be used to recover the document. This is a great idea. However, even the other MS Office applications do not implement this feature. If this was a modular system, one module could be a universal "data storage" module, and only this one module would need to support auto-saving. A user could even decide to place these temporary files on two different drives, or even place them in a location online, to increase security even more. Any application could then use this module, and any document-editing application would then have this feature, and all options set by the user would (or could be) universal across the entire system.

Constructability goes even farther, however. Suppose we have some sort of instant messaging program. We also have a simple encryption module, using public key encryption. This encryption module could be placed

"in between" the user interface object, which would collect the input, and the network protocol. When text was typed, it would first pass through the encryption module, which would encrypt the text using the partner's public key. This would be passed to the instant messaging network module, which would send the encrypted text just as it would send any other text. The receiving computer would also have an encryption module in the same location, and when an instant message was received, it would first be decrypted by the decryption module using the private key, and then sent to the user interface. As you can imagine, transparent encryption of email would be just as easy to achieve. This modularity allows for a level of control and power over the computer which is unheard of today, except by those who are able to program proficiently and know the APIs well. Even then, implementing an encryption scheme such as the one described onto a currently existing messaging program would be a "hack" at best, and horribly unstable at worst, unless the entire instant messaging system was open source. Obviously, increasing constructability increases efficiency, both for the user, who can now do more in less time, and the programmer, who spends less (or theoretically no) time creating the same routine multiple times. The encryption scheme above would clearly increase security as well, and it could essentially be implemented into any network communication system.

Portability

Modularity is also a great advantage in maximizing portability. Due to the fact that only a few modules directly communicate with hardware, all others are essentially platform independent. And the goal is to be able to distribute software which can run on any platform. But there is still a major problem in regards to coding. There are currently two options when it comes to software distribution. If one distributes the compiled program, it can only run on one CPU family, and generally only one type of operating system architecture. If one distributes the source code, as is done in Linux, the workings of the entire program is open for inspection and alteration by any party, and can generally be compiled to any platform. While this is definitely an advantage for free software, companies are understandably wary of distributing source code. The solution that The Force chooses is to create a virtual "mid-level language." High level languages would compile down to this language, and this could be distributed. The MLL code could then be compiled on any platform into it's own machine code, or interpreted, as Java currently is. The MLL language would be low-level enough so that it would be as difficult to understand by inspection as assembly language, but abstract enough so that it would not rely on any specific word-size, instruction collection, or processor architecture. Java may be a possible MLL, but the fact that it is proprietary limits its usefulness, and it has certain limitations that may make it difficult to use for the Force (such as a set word size). Designing the MLL language represents the most technically challenging aspect of The Force project.

Capitalism or Communism?

The Force concept of software design is very compatible with a free source system of development. However, it is also very well-suited to a capitalistic market as well. In fact, modularity allows for *true competition* among software developers, as opposed to the monopolies in place now. The best module for a specific task rises to the top, as opposed to the current system, where people are limited by what is "compatible" with their previous software, or what the majority of people are using. A modular system would allow one to use the best module for each different task, while still retaining compatibility with other users and files.

Status and Future Directions

Currently, there has been no coding done for the project. I have come to the tentative conclusion that the most efficient and least tedious way to start would be to develop a Force interpreter on top of Linux. This would allow use of a broad driver and software base, and would start with a stable system. The MLL language could

be developed here, and many essential modules could be developed. These could, if designed correctly, then be used in a full implementation of the operating system, and be compiled into machine code modules. A similar interpreter could be implemented in the Windows platform, however, due to the fact that windows is copyrighted and not open source, it would be technically difficult and illegal to use parts of the OS in a full Force implementation. Therefore, the quickest path to a complete OS appears to be through Linux.

The first step is to begin the specifications of the Mid-Level Language, and this has begun. One major problem when creating a cross platform language is the word size of the processor. This is indeed a problem for C/C++ applications, as one compiler may use an integer of 16 bits, and another may use 32 bits. The proposed solution to this is to have applications specifically declare the range of variables. This is also a way to seamlessly integrate range checking, without any extra programming. A programmer could declare an integer of, say at least -32,000 to +32,000. Declaring this way would allow a compiler to use a larger datum if it was more efficient. An integer could also be declared more strictly, and if the platform did not have any such datum, it would compile in dynamic range checking to enforce this range. Clearly the former strategy allows for faster performance, while the latter allows for increased robustness. More information on the current MLL design will be posted soon.

Want to help?

Any and all types of help available is greatly appreciated. Please send an email to jonathan.jaekle@uconn.edu with your interests, knowledge, and skills. Specifically needed are people who know Linux APIs, Windows APIs, various CPU architectures (most specifically x86, PowerPC, and other RISC CPUs such as Strong-ARM). Programmers with a broad knowledge of languages are also needed. People who have knowledge of any high level area of computing will also be needed once the MLL specification has been laid down, as there are hundreds of modules that need to be made. All software which I produce or collaborate on in this project will be open-source, and under the GNU public license. However, even if other people will not distribute their source code, they will be strongly encouraged to make modules, as simply the existence of quality modules will help to development and proliferation of the system.

If there is interest in the project, I will start a message board to facilitate communication among developers. I intend this project, if people join, to be a very "democratic" one, with no central administration, unless for some reason it becomes needed. However, those who have disagreements which they are not willing to compromise on are encouraged simply to make modules as a proof of concept. Therein lies the strength of the module system. People who have a solid understanding of website design and web programming are also needed. Please contact me at jonathan.jaekle@uconn.edu. My AIM screen-name is xthexjackalx .

Comments?

Please contact me with any feedback, positive or negative, at jonathan.jaekle@uconn.edu. My AIM screen-name is xthexjackalx . Feel free to contact me in either way, as I would be interested to have a real time discussion.