# About Software

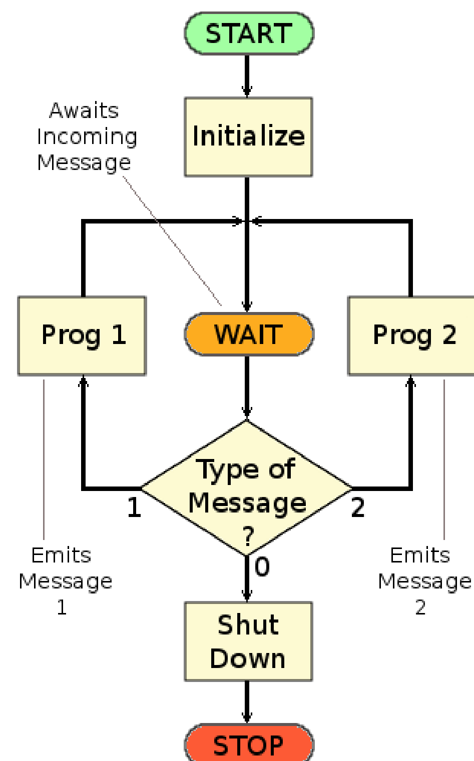
*To write a good robust software package, that serves the best interests of its users, one must follow certain rules. But to write a commercially successful software package, which maximizes the profit of its producer, one must follow a very different set of rules.*

To write a good robust software package, that serves the best interests of its users, first analyse carefully the task that the package is required to fulfil. The relevant activities of the human users form part of this analysis. From this analysis, construct some form of logical model of the method by which the package will fulfil its task. Then iteratively reduce the logical model to its simplest form, excluding any superfluous functionality, which may have hung over from the manual methodology traditionally used to perform the required task.

## Create a Logical Model

There are many established kinds of logical model up on which a software package may be constructed. For most of the projects on which I have worked, the most appropriate kind of model has been what is known as the *finite-state machine*. In each of these cases, the *finite-state machine* was made to work by receiving and sending messages from and to the user. The logical model was thus what is called a *message-driven finite-state machine*, which is generally referred to more concisely as a *finite-message machine* or FMM.

The logical essence of an FMM is shown on the right. It would probably be written as a dæmon, starting automatically when the operating system is started, or being started manually by a terminal command. At start-up, it does its initialization procedure and then goes to its WAIT state. Here it listens on a designated calling port for messages from human users. It responds to only three kinds of messages, namely: "shut down", "execute Program 1" and "execute Program 2". If told to shut down, it does its shut-down procedure and stops. If told to do either of its two types of task, it runs the appropriate program and then returns to its WAIT state. Each program, once it has completed the task requested by the user, sends a response message. The FMM could be made to invoke a new instance of the appropriate program to handle each arriving message. Alternatively, it could always use the one instance and queue incoming messages, waiting each time for the program to terminate before dealing with the next message in the queue.




The chosen form of implementation of the logical model will depend not only on the nature of the user task, but also on the type of computer and operating system selected. At this stage, the logical model of the user application must be rigorously documented in such a way that the eventual users will be easily able to understand it. This part takes a lot of skill and patience.

## Code-up The Programs

Next, select an appropriate programming language (such as C or Java) and code up the various message-handling programs which make up the package.

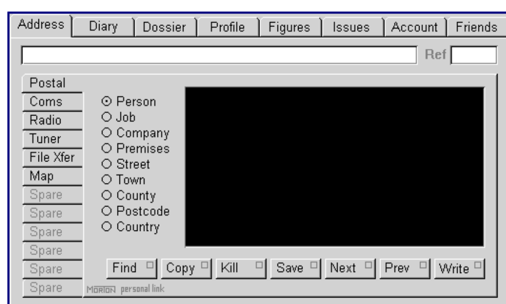
For example, each of the FMM's message-handling programs could be written in C in the form of a Unix command. A Unix command is like an imperative sentence written in English. It comprises a subject, verb, possibly a verbal qualification and an object, as formalized in the following table.

| subject | verb | qualification | object |
|---|---|---|---|
| user | program | options | argument(s) |
| rob@neddy | edit-customer | address-only | customer123 "15 Bridge Street..." |

The equivalents of the English parts of speech, as they appear in a Unix command are shown on the lower line of the table. In normal circumstances, the identification of the user does not need to be explicitly specified. It is implicit because he is whoever is signed into the terminal at the time. His identity is shown already by the operating system before the entry prompt. However, for a complete FMM message, which could have come from any one of multiple users, it is necessary to specify the user explicitly, as it is in an English sentence when the subject is not the speaker. In the example above, the user is a human called "rob" who is currently logged in on a computer whose network name is "neddy". Rob has requested a change to a customer's address details, giving the modified address as part of his request.

If the user be on a different computer from the one hosting the FMM, then the command-line message above must be encapsulated within an IP packet in order for it to be passed across the network that links the two computers. The FMM, on receiving the IP packet, would then strip out the internal command message and pass it to the Unix shell to invoke the appropriate message handling procedure (Program 1 or Program 2 in the above example). Of course, the FMM in a real application package would most likely have quite a large number of message-handling programs.

## Design The User Interface

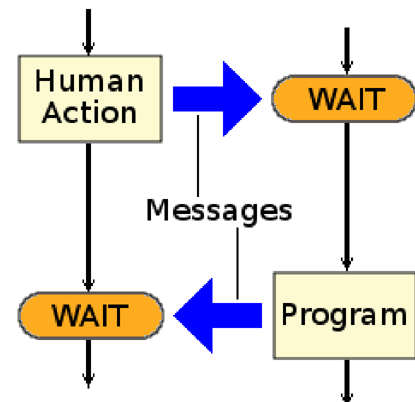



The next stage is to create an electro-physical interface through which the human users may send messages to and receive messages from the logical model within the computer. Nowadays, this is normally a client-side program, which constructs and operates a graphical user interface, an example of which is shown on the left.

In his day-to-day work, the user's focus is on his business: not on your system. For him, the computer and its software are merely a tool. For this reason, the GUI should be to him as transparent, unimposing and self-intuitive as possible. Consequently, when designing the window or frame layout of the GUI, don't indulge your graphic design skills by creating needless eye-catching glitz and fancy unconventional widgets. This will serve only to distract the user from his business task, make the GUI more difficult for him to use and slow him down in his work. It will also make the GUI much slower to learn for new users. Keep the design of the GUI as simple and conventional as possible. Make labels and text clear and easy to read. Use only standard widgets: buttons, fields, links, menus and tabs, which users can intuitively recognise.

The human interface does not necessarily have to be graphical. In a lot of cases an old-fashioned text-based interface is more appropriate; being simpler, faster, quicker to learn and easier to use. Through the interface, the information the user wishes to see must be presented in a form that is most easily digestible by the user. This is not necessarily the form in which it is most convenient to retrieve the required information from the logical model.

A user probably already knows what he needs to do to fulfil the day-to-day requirements of his job. What he probably does not inherently know is how to do this specifically through the [new] user interface. It is therefore essential to include, as part of the user interface, procedural instructions for each possible interaction that may take place between the human user, sitting in front of his screen, and the logical model hidden behind it. The procedural instructions should have links to background information to give the user the broader context he will need to fully understand what is going on.



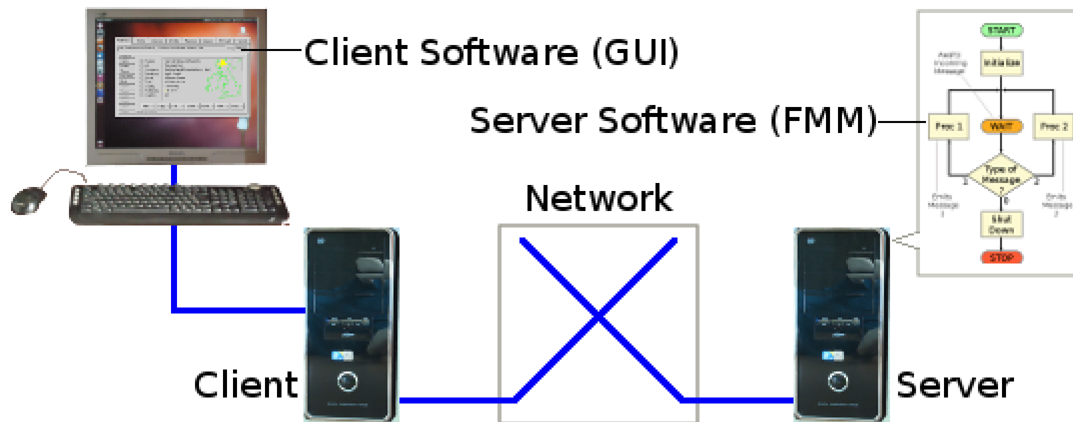

There are many different ways of providing this user documentation. The first is conventional paper books. Another is to write the documentation as a set of interlinked HTML pages on a CD or hard drive. A rather novel option is to implement the user interface as a Java applet embedded within its own HTML documentation.

My preferred option, however, is to implement the user documentation as a set of context-sensitive help texts. In this case, when the user clicks the HELP button on the screen, a window frame opens in which appears explanatory text pertaining specifically to the step in the program with which he is currently involved. These help texts must be carefully crafted and iteratively improved in the light of difficulties experienced by users. The context-sensitive option saves the user having to go and find the information he needs in a book or file reader. For this reason, I think it is the best option for minimising user stress and fatigue.

## Typical Implementation

Most software packages nowadays are expected to be able to provide service to users spread over many computers. These may be located not only in different parts of an office or factory, but also in different parts of a country or even the world. The FMM part of the package must therefore run within a central server, while many copies of the user GUI part of the package must run within many different client computers in different locations.

Below is shown the server computer running the FMM Server Software with just one of the many possible client computers connected to it via a network. The network can be either a local area network or the Internet itself.




The user enters his requests to the FMM through the package's GUI, as shown on the screen in the above diagram. The client software, operating behind the GUI on the user's client computer, converts his request into the form of a command-line, encapsulates it in a network packet, and sends it to the FMM running in the server. The FMM in the server then executes the requested command and returns the results as a message sent back to the client software of the user concerned. The GUI software then displays the results for the user to see on the screen.

---

A well-designed software package, thus constructed and implemented, should be self-teaching to reasonably self-motivated new users and should rapidly become maintenance-free.

From my experience over the 50 years since I began programming computers, producing software of this kind, and in this way, will certainly provide you with satisfied users. But it is definitely <u>not</u> a recipe for commercial success.

## Opting For Commercial Success

To write a commercially-successful software package, which maximizes the profit of its producer, one must already have, or be able to acquire, rather a lot of capital. What follows involves initially spending an awful lot of money.

First survey the market. Conduct surveys to find out what the most influential set of potential customers actually want. Be aware that what they want is what they *think* they need. This is not necessarily - in fact, it rarely is - what is actually best for them.

The results will be a mess. The surveyed potential customer is not usually an expert systems analyst. He will probably have his own cloistered traditional way of doing the task concerned, which will have just "grow'd" like Topsy as his business evolved. Consequently, this way (or methodology) is unlikely to be very efficient. It probably contains procedures that "go round the sun to meet the moon". But it is what the customer wants. So it is what the customer must have.

Create an amalgam of the survey results for all the potential customers whom you surveyed. Thus you have a market view of what the task is and how it should be done. The amalgam is guaranteed to be even more of a mess than the singular view of any one of the survey participants. Notwithstanding, you must use this as the basis of the *il*logical model from which you will construct a software package to fulfil the task concerned for the industry you surveyed.

You now have the specification of a software package, which conforms to the wishes and desires of the generic customer in the market-sector you surveyed.

## The Need For a Glitzy GUI

Now to write the software. Never mind about the underlying functionality that must take place within the server. Begin with the GUI. Design a user interface that displays what the generic customer wants to see and how he wishes it to be presented. This is unlikely to be all the information, and nothing but the information, that he really needs. It probably contains superfluous information he does not really need and lacks other details that would be useful and could even eliminate the need for one or more other displays. But it is what the generic customer has asked for.

Remember that you are building a software package for sale. You therefore need to impress the potential buyer. The buyer is rarely the user. The buyer is who has the money, such as a company executive. The user is generally a lowly employee. You don't need to impress him. The majority of buyers - the generic buyer - is not technically knowledgeable. He can therefore be impressed only by what he can actually see. For this reason, you must make your package's GUI appealing to look at. It must be a graphical work of art, with suave colour gradients, delicately sculptured buttons and a (possibly animated) corporate logo, which exudes precision, excellence and commercial confidence.

Fancy eye-catching presentation is absolutely essential. It must be forever new and different. Just like a fashion show. This is what the buyer has grown to expect. At software fares and business exhibitions, what else can the ignorant generic buyer go by, as his aching feet tread their way around the endless stands, glancing at the jazzy screen displays through the thick leaves of the ubiquitous rubber plants? Of course, these jazzy displays - with their glowing colours, idyllic images and wriggling applets - take forever and a day to load or update. But who cares? He, the buyer, does not have to suffer the daily frustration of their wobbling procrastination. That's for the poor user to endure for ever after. He's the one who has to spend his time waiting upon the sluggish GUI, straining his eyes trying to read the merged r's and n's of its immaculately set microscopic type. He's the one who has to play the game of "guess where to click" within the unconventional graphical confusion on his screen.

Next, create a menu or tab system within the GUI to accommodate all the actions the generic customer wishes to do in order to manipulate his information in the way he wants. Minimise the use of words [text]. Instead, represent as much as possible using icons: little stylised pictures that represent objects and actions. Finally, get your programmers to rapidly cobble together a server-based handling program for each action. It doesn't matter if it's full of bugs. What is important is to get it out of the door and into the market as quickly as possible. You can straighten out all the problems later through your on-line automatic update facility.

## The Training Gravy Train

Unfortunately, what action appears in which menu or sub-menu depends on the methodology that evolved within the generic business of the generic customer within the industry concerned. Be aware, however, that no one real customer is the generic customer. Each user will be accustomed to doing things somewhat differently. So no real user will be familiar with doing things the generic way.

Furthermore, the generic methodology will inevitably be rather a jumbled mixture that is anything but logically optimised. Consequently, no user will be able to work out how to use the GUI simply through common sense. He will have to be specifically taught how to use the GUI, *learning by rote* what to do in each situation. The copious use of icons adds the necessity for the user to be taught an entirely new over-simplistic grammarless non-intuitive sub-language for each application, which adds to the amount of training required.

To be the producer of a commercially-successful software package, this is just what you want: a package whose GUI looks like an over-dressed Christmas tree, which is distracting and difficult to

get the hang of. Why? because it creates a grand business opportunity to sell training to customers who, having bought your package, cannot use it without training.

## The Consumer Market

With the advent of tablet computers and the smartphone, with their extremely small screens, the GUI has had to be made ever more simplified and compact. This has driven the tendency towards all-icon GUIs.

Icons constitute a sub-language that is separate and distinct from a natural spoken and written language. Icons are not standardised. What an icon means depends on its context. Different software has its own set of icons. Hence, a user has to learn what each icon means within each different application program. Consequently, although an iconic sub-language may appear simple to look at, it has to be separately learned.

Icons are easy for pre-school children to grasp. Indeed, as I understand it, they were created within the context of the way pre-reading age children think. As a result, their scope is very limited. But, as St. Paul said in 1 Corinthians 13.11:

> *When I was a child, I spake as a child, I understood as a child, I thought as a child: but when I became a man, I put away childish things.*

The thinking processes of adults are very different from those of pre-reading age children. Icons do not serve well in complex tasks, and if made to do so, become acceleratingly cumbersome.

GUI users are thereby becoming increasingly forced to do things the way a pre-reading age child would. It is like the television sound bite and the small-block newspaper ad. It reduces the adult's attention span and scope of thought to those of a child. It thereby erodes one's ability and propensity to think as an adult.

Whether intentional or otherwise, the way in which the graphics user interface has developed, especially for small personal devices, has tranquillised the adult mind, focussing it onto banality and away from serious thought. Hence the endless truck-loads of trivia that traverse the Internet daily via email and social media. Perhaps this is what the *powers that be* want: a doped, placid - and hence easily controlled - populace.

## Advertise, Advertise, Advertise

All the while, pour loads of your initial capital into advertising your software as being "easy to use". Don't worry. Since your software is destined to dominate the market, it will be the only software that most will ever see, and the stupid consumer will never be any the wiser.

Place a lot of your advertising in newspapers and specialised industry magazines, thereby obligating their respective journalists to write glowing reports in articles that endorse your software, while issuing unfounded warnings to the hapless buyer against purchasing the products of "unknown" small-fry.

Now you are well on your way to dominating the software market. However, by taking a few dubious extra steps, you can make your position in the market pretty well invincible, while, at the same time, creating further opportunities that will enable you to really rake in the money.

## The Virus And The Trojan

Covertly commission various renegade software experts to create programs known as viruses and Trojan horses. These are programs which, by secretly installing themselves within people's computers, can cause them to run painfully slowly and perhaps even corrupt or destroy people's personal data files. Creating these programs requires considerable technical knowledge, skills and resources, so it is necessary to bankroll their creators. However, to avoid any suspicion, spread the illusion among the general public that these programs are easily-written nuisances cobbled together by rebel hackers and schoolboy pranksters.

Again covertly, distribute these viruses and Trojans throughout the Internet. This is best done by coming to some arrangement with the operators of certain web sites, which will contain ActiveX controls that will download the viruses and Trojans automatically into a person's computer whenever that person visits the site. The safest kind of website to use for this task is the porn site. This will greatly discourage people from trying to track where the virus or Trojan came from because the victims will not want to admit to accessing sites for the purpose of viewing dirty pictures.

You have successfully created a real and present fear within the market. The market needs a saviour. And whoever would be the saviour shall, in the execution of his act of salvation, stand to make a lot of money. You therefore commission somebody to write what shall become known as anti-virus software, whose task is to hunt down and destroy any and all viruses and Trojans that may be infecting a person's computer. To avoid any suspicion of collusion, you and the anti-virus software producer agree that he shall present himself to the public as a completely independent business interest. Driven by the fear of losing personal data, and frustrated by sluggish computer performance, the public rushes out to buy the anti-virus software.

In the meantime, keep your renegade software experts working hell for leather to keep producing better and ever different viruses and Trojans to continually circumvent and thwart the defences provided by the anti-virus software. Likewise, the anti-virus producer keeps producing and selling a continuous stream of updates to combat the ever-mutating viruses and constantly evolving Trojans. And so the wheel turns. And so the money keeps rolling in.

## Lock Out The Small-Fry

Next, completely lock out of the market all but yourself and other members of the global clique of the big software corporations. Make sure that the dedicated hard-working highly-skilled independent software artisans, the world over, never get a look-in. Do this by popping up a warning message whenever a person tries to install any "unknown" software from any "unknown" supplier, implying that the software concerned could be "malware".

Malware is a derogatory catch-all term that includes viruses, Trojans, and other software that is presumed, without basis, to be potentially harmful to people's computers. It also includes "spyware": a legitimate looking application program, which contains extra undeclared functionality. Such functionality could, for instance, read a person's private files on his hard drive, surreptitiously passing any information of interest to an unknown remote third party across the Internet.

The warning message pops up whenever a person tries to install any software whose producer has not paid the high price for having his software certified, by the global clique, in order to obtain a digital certificate, which would allow it to be installed without the warning message appearing.

## The Beckoning Back Door

Now for your *pièce de résistance*. To keep your quick out-of-the-door bug-ridden software working acceptably well, you need to be able to update it very regularly. Whether your software be an application package or an operating system, you need an automatic update facility. This works behind the scenes on each user's computer. The user is generally unaware of when it is operating or what it is doing. To be able to work, however, it needs full administrator access to all resources within the user's computer. It must therefore be able to read and write to any part of the user's hard drive.

This presents you with rather a tempting and potentially rewarding opportunity. You build, into your automatic update facility, a back door to every user's computer. Through this back door, you access any and all the personal information the user has stored on his hard drive. And you do this without him ever being aware of what you are doing, or that you are doing anything at all. From this personal information, you construct a consumer profile of each user. You then sell a vast number of these profiles, for an extremely good price, to commercial interests who subsequently use them to target people more precisely with their product advertising and promotions. This is probably your greatest money-spinner: the one thing that positively guarantees your commercial success.

But there is a far more sinister use to which this back door may be put. It can be used by government agencies to expedite the most invasive, yet completely stealthy, form of spying upon any chosen individual. A legal instrument could be issued, forcing you to give them access to the world's back doors. Criminals too, if they could compromise one of your relevant employees, could also gain access to any and everybody's back door. That should make everybody feel uncomfortable. Not that he may have done anything wrong. Personal information on one's hard drive, when taken out of context, can easily be misinterpreted and misunderstood.

As well as having an operating system, a personal computer has many application programs installed, each with its own automatic update facility running in the background. This potentially provides as many different vendors with as many back doors into the user's personal files. The opportunity is there. It is so easy to implement. It can operate unseen. Nobody will know. Is it therefore realistic to suppose that all software producers would, through moral virtue, wilfully refrain from engaging in such skulduggery?

## Conclusion

So, how can the poor lowly user be protected? How can he gain real security? The only solution, in my opinion, is open-source software. With open-source operating systems and application programs, any one of millions of independent programmers, all over the world, can examine whichever part of an operating system or application program he wishes and thereby know exactly what it is doing.

Of these millions, there will always be somebody who decides to get his claws into every last piece of code and who will blow his worldwide whistle on any program that is doing something against the best interests of its users.

---

© Feb 2014, May 2017 Robert John Morton | Related article: Who Owns Cyberspace?

©This content is free and may be reproduced unmodified in its entirety, including all headers and footers, or as “fair usage” quotations that are attributed as follows: “ - [article name] by Robert John Morton http://robmorton.20m.com/”