Gotos Considered Harmful and Other Programmers' Taboos

Lindsay Marshall
James Webber

Department of Computing Science,
University of Newcastle upon Tyne
Newcastle NE1 7RU, UK
{Lindsay.Marshall, James.Webber}@newcastle.ac.uk

Abstract

Programmers are constrained not just by conscious application of rules and procedures, but also by taboos that they have acquired as part of their formal education or informally from colleagues. These taboos usually embody perceived sound advice and have generally been concerned with the breaking of abstraction boundaries. However their effect can be to needlessly restrict the range of solutions to design problems that programmers consider. This paper examines a set of common programming taboos, and addresses both social aspects and technical reasons as to why programming taboos have arisen.

Introduction

Every social group has its taboos, though the processes for establishing and lifting them may be more explicit in some cultures than others. Thus, as you would expect, programmers, have their own taboos – though these are not usually labelled as such by the community. This paper looks briefly at some of these taboos, how they originated and developed, and how they have been internalised as part of programmer culture. As teachers (and students) of Computing Science and observers of other such teachers, we have seen and taken a part in the process of passing on, and indeed reinforcing, some of these taboos. In addition we shall look at some of what we believe to be "taboos in the making" which show that the process is always with us.

First, it is important to state what we mean by a taboo. Chambers defines the word to mean "a system of prohibitions connected with things considered holy or unclean," and this captures almost exactly the nature of this discussion, mainly the unclean, but also the holy. The only quibble with the definition is with its use of the word "system" which perhaps implies a more formal arrangement than actually exists in the world of the programmer. They key point about taboos is that the prohibitions are frequently so strong that people are simply not aware of the influence they have on their day to day behaviour.

At this point it should be stressed that, as in any other society, various groupings will respect taboos either not at all or in different ways. All of the comments presented here are generalisations and it is simple enough to find counter examples. We do not believe that this invalidates our conclusions about the presence of, and reasons for, the taboos mentioned.

Four taboos

Explicit Control Transfer

One of the longest standing taboos amongst programmers is against the use of explicit control transfers, that is the goto statement. The principle source of this is the letter to the editor of the CACM "Go To Statement Considered Harmful" by Dijkstra (1968) and from there the structured programming movement of which the paper was part of the spearhead. In the decade after this paper, most formal teaching of computing emphasised the dangerous nature of the goto statement and penalised its use where other constructs could replace it. Languages started to appear in which there were no goto statements and no way of labelling specific lines of code.

Once this trend was established, and, at least in academic computing, structured languages were the norm, increasingly less mention was made of the goto statement. Today, students learning programming languages are not normally aware that the possibility of such unstructured control transfer exists. Goto-heavy languages such as FORTRAN IV and BASIC are considered "unclean" and Dijkstra is considered "holy."

Of course, we do not wish to imply by any of this that the introduction of structured programming was a bad idea. It was not – the whole act of goto-avoidance provided motivation for the development of the structures that help make the construction of today's complex programs easier. Loops, blocks and if/then/else structures easily conquered the explicit jump for normal control flow, but it has taken much longer for there to be an adequate provision of mechanisms in the area of exceptional flow, the last place had frequent use. Note that Java has **goto** as one of its reserved words, but does not actually make use of it.

Where the problem lies, as with all taboos, is in the loss of knowledge and experience. If something is forbidden for long enough, it becomes difficult to resurrect the knowledge of how to use it. The programmers being produced today, if faced with a language that needed the use of explicit control flow would find it hard to adapt to the freedoms it allowed and disciplines that it required. This leads naturally to the next considered taboo.

Low Level Programming

Almost no-one programs in assembly language today. Compilers generate code that is as good and often better than could be written by hand, and RISC architectures make hand coding not only tedious but also very intricate. Students learn less and less about the specific architectural features of machines such as registers, condition codes, addressing modes and other esoterica. Few programmers now would ever dream of rewriting a piece of critical code in assembler to improve speed or to take advantage of a particular machine feature (assuming that there was speed to be gained or that the machine had any special features!). But someone has to understand the order codes of machines in order to write the compilers in the first place and to write the bootstrapping code needed to get everything else working.

Attitudes to assembler programming are an interesting mix of the holy and the unclean. If you can actually write assembler and understand what is happening when presented with a piece of code, then you will impress others with your skills. Just as used to happen (and probably still does) if you could read raw, hex coredumps. You will definitely be seen as a guru. At the same time, these admirers would never dream of writing any assembly code – it's dirty, difficult, dangerous and discouraged. For epopts only.

Skill loss is once again the problem. Effective low-level programming is hard and having unskilled, or at least, unpractised, programmers doing it is problematic. People who are going to be systems programmers ought to be given some experience of low-level programming, if only to allay their fears about it and to show them that it can be an acceptable solution in some circumstances.

Flowcharts

The third great programmers' taboo is the use of flowcharts. In the heyday of unstructured programming languages flowcharts were everywhere. Programmers all owned, or aspired to own, a flowchart stencil. Flowcharts appeared whenever people wanted to indicate programming activity in cartoons or films. And then they were banished; almost overnight it feels, though the process was certainly slower than that. Why did this happen? Did structured programming, better commenting practices, and the use of pseudo-code descriptions all combine to make the flowchart seem old-fashioned, a legacy from the bad old days of gotos and assembler with which they are inextricably bound? Or was there some other reason? It is difficult to say, given the length of the elapsed time.

Flowchart stencils languished at the back of drawers and people tried to write program specifications in formal notations, or used block diagrams to show structure. Very occasionally someone might use a diamond shape to indicate a decision. But nobody drew a flowchart. Ever.

But it seems that using diagrams to explain is a fundamental part of human nature. Flowcharts are coming back. Not, we hasten to add, called flowcharts, a name which is still most definitely taboo, but referred to as design notations, structure diagrams, or even just as "sticks and boxes"." The rise of object-oriented programming has emphasised the need for programmers to be able to picture the relationships between objects and so design notations have flourished. The most successful today is of course UML which is vastly more complicated than any flowcharting system ever was with nine different main types of diagram. So complicated, in fact, that many people are at a loss to know how to use it effectively and end up drawing what are essentially flowcharts with it, though, of course, they are never called that. What makes UML clean rather than unclean is that it has formal underpinnings, even though these are of no real interest to most users of the system.

Global Variables

Global variables are bad and should not be used. Standard advice to the novice programmer these days. And, once again, not wrong. It is better to pass parameters and use data structures to wrap up related data than to have lots of individual, global variables lying around in your program's address space. This taboo comes mainly from experience with the horrors of FORTRAN Common blocks where unconstrained lists of variables were mapped on to the same address range, with all the possibilities for error and confusion that this can cause.

This increasing tendency to structure both data and control has lead, inevitably, to object-oriented programming and all that entails. And interestingly, one of the things that OO programming entails is that it is often necessary to have global variables. Certainly these are global *objects* rather than, say, a single integer, but they are still global. The notion of a persistent object store implies an address space filled with objects, all of them residing at the same conceptual level (though they may be nested inside namespaces which give some structure to the space). The dynamic nature of many programs means that they have to locate objects at runtime and to do

this, there needs to be at least one global object – the one that provides the location service.

Another area where global variables are increasingly used, is in event driven programming. An event fires, carries out its task and leaves its result in a well-known place for other event handlers to find. Students coming to event driven systems after learning straightforward sequential programming find the use of global variables one of the hardest things to get to grips with, largely because they have been taught to write result returning functions with parameters which are called by other parts of their program. The idea of using a common place to store values is somehow alien to them, though it should be noted that, strangely, they find no difficulty in understanding the relationship of the file system with a program which essentially provides exactly that. Once again, a taboo leads to a difficulty in skill transfer.

Taboos in the making

The reader should by now have their own ideas of what they would regard as existing taboos amongst programmers and probably can suggest several areas where taboos are being created at the moment. Let us suggest three that we think Computing Science education and practice look at as increasingly unclean.

Explicit addressing

Pointers, like assembler programming, are dirty, difficult, dangerous and discouraged. In fact they seem to hark back to the kinds of things that assembler programmers got up to with explicit values and address registers. Academic courses often do not introduce students to the use of pointers till quite late on in their course (if they do so at all) and even then treat them as too hot to handle. One of the great benefits touted for Java is the lack of access to pointers and C has never been a popular teaching language, at least in Computing Science departments, partly because of its use of pointers.

All the same arguments we have seen above apply here. Pointers can and do lead the unwary into trouble and it is better if to manage without them if possible. However, there are important programming techniques relying on the use of pointers which students do not know about. You cannot write a Java program to drive real hardware because you cannot (without hackery) talk to the explicit addresses that make devices work. Without a knowledge of pointer addressing it can be hard to understand how arrays are stored, especially multi-dimensional arrays. For systems programmers this kind of knowledge is important.

There are of course still thousands of lines of code being written in C and C++ using pointers every day, but voices against their use are growing louder all the time and talking to students and reading their code, you can see the taboo against them starting to form.

Data Formats

Imagine suggesting to a prospective client that you intended to store the year in a two digit field in a program you are writing for them. Even if this is a perfectly reasonable thing to do in the circumstances, the background of Y2K means that nobody would entertain your suggestion for even a minute. Two digit dates are not a future taboo, they are taboo now. But note that other areas of date confusion (leap year algorithms, month/day ordering etc.) have not been resolved by this hyper-awareness of potential date problems, even though they have always been a problem.

There are other data format taboos appearing, though none as rigid as that for year representation. Storing data in binary, even when this is the best way to do it, has been becoming increasingly uncommon for several years now, and becoming more common is the idea that you should store your data in a structured format using something like XML so that it is easier to process in different ways. And, once more, there is nothing wrong with this, except that for many applications it is simply inappropriate and the effort involved in producing a proper DTD for the data format and all the tagging that may be needed far outweighs any benefits that will be obtained in the long run. But with the webification of computing this "structured, flexible data" trend is set to continue.

Real machines

Even as the number of real, hardware architectures in everyday use decreases, so does the pressure to move away from any specific architectures at all increases. Java with its virtual machine, and higher level, interpreted languages like Perl, Python and tcl/tk take programmers further and further from the machine. Add to this support environments that conceal the details of linking and loading, and you end up with programmers who do not have an adequate working knowledge of what goes into the executables of their programs. Not a problem until they have to debug them or even just understand error messages.

What is strange about this situation is that there is nothing new in any of these technologies. We have had interpreted languages with or without virtual machines for many years and they always co-existed with native systems. Why has the idea suddenly taken hold that machine independence is such a good idea? (Which it is, up to a point.) The idea of writing portable software has always been encouraged, but it never seems to have acquired taboo status, but Java fans tend to regard the machine independence as a holy characteristic of the language.

Conclusions

Looking at the various taboos described above, there are two patterns that emerge. The first is clearly a social one – programmers pass their prejudices to other programmers, particularly teachers to students. This probably explains the demise of flowcharts, the blessing of open source and the ubiquitous hatred of Microsoft. Fashion certainly has something to do with it – the only trouble with fashion in programming is that you can end up with the equivalent of patching those old, flared, brushed denim loons for the next ten years, long after everyone else has stopped wearing them.

The second factor is more interesting. If you look at each of the instances above you will see that they share a common feature – all of the taboos are concerned with the crossing of abstraction boundaries. Between explicit and implicit control flow, low-level and high-level coding, local data and global data. People are loathe to cross boundaries, either for reasons of safety or for lack of skill. And it is the latter that makes taboos dangerous. As we have shown, the introduction of a taboo can cause knowledge and experience to be lost. Knowledge and experience which still have their uses – we are not talking about skills that have no market.

In addition, even if these skills were no longer particularly useful, the crossing of boundaries is an important part of the design process. If you always design within the same constraints then nothing new ever comes along. A point in case is GUI design – everything looks like a web browser now. Taboos stop people from crossing boundaries and they do this without anyone realising it, which is why taboos are more

dangerous than simple rules. By becoming aware of these taboos programmers can learn when they are subconsciously following them and perhaps see new, different, possibly even better solutions to the problems they encounter.

Acknowledgements

Thanks to Paola Kathuria for comments and support.

Reference

Dijkstra, E.W. (1968) Go To Statement Considered Harmful. *Communications of the ACM.*, 11(3) 147-148