Going, going,

As programming evolves and becomes easier, the programmer of tomorrow may essentially be a non-programmer

In the glory days of programming, tightly written, efficient code was sacrosanct for programmers. Hand-coding was God, and programmers were believers. Today, programming is a dying art. After all, when reusable chunks of pre-written code can be patched together to create an application, who wants to 'program'?

And if applications, created by people who are essentially non-programmers, are bloated beyond all reasonable limits, who cares? A new processor is always ready and waiting to happily run the bloatware that these programmers churn out.

Have we reached the point of no-return?

The evolution

By writing all those mysterious lines of code, ideally, in as few 'words' as possible, the programmer tells a computer what to do. The computer cannot understand the English-like instructions so the instructions have to be 'compiled' into binary numbers—zeroes and ones—by a what is called a 'compiler' so that the computer can run the resulting program. Then there are interpreted languages (like BASIC), where the computer translates programming instructions as it runs the program. This is slower, but easier for programmers—they can, if required, make changes to the code, and run the program instantly, without having to compile it again. That was, basically, the way things worked until a couple of years ago.

Now, however, developments in technology have changed the way programmers program. Terms like RAD (Rapid Application Development), OOP (Object Oriented Programming) and Visual programming, have entered the tech-savvy programmer's vocabulary. The buzzwords might vary, but the goal remains the same—to make programming easier for those who are programmers, and accessible to those who are not. This would let the programmer concentrate on

The Evolution of Programming

- 1949 - Short Code, the first computer language to be used on an electronic computing device, makes an appearance.
- 1952 - Alick Glennie, working in the University of Manchester, designs a very basic programming system which he calls Autocode. It is a compiler with limited capabilities.
- 1957 - FORTRAN (mathematical FORmula TRANslation) system is released. John Backus, as head of the team which developed this language, garners wide acclaim. FORTRAN II appears in 1958, and FORTRAN IV in 1962.
- 1958 - John McCarthy, at MIT, begins work on LISP (LISt Processing). LISP 1.5 is released in 1959.
- 1959 - COBOL (Common Business Oriented Language) is created by the Conference on Data Systems and Languages.
- 1960 - ALGOL 60, the first block-structured language, appears, and goes on to become the most popular programming language in Europe in the mid-to-late 1960s.
- 1964 - At Dartmouth University, professors John J. Kemeny and Thomas E. Kurtz invent BASIC (Beginners All Purpose Symbolic Instruction Code). The first BASIC program is run on 1st May, 1964.
- 1968 - Niklaus Wirth begins work on Pascal. The first implementation of Pascal appears on a CDC 6000 series computer in 1970. The Pascal User Manual and Report is published in 1975, and is considered to be the definitive reference on Pascal to this day.

almost gone!

solving a problem without having to scale a steep learning curve.

RADical change

HyperCard (a forerunner to Visual Basic) popularised the visual-prototyping style of programming when it made an attempt to represent every program function as an on-screen object. Visual programming implies an object-oriented, or at least an object-like view of a program. It entails the visualisation of the application design from the point of view of user-interface functionality. OOP and RAD methods have taken this concept to a higher level. RAD tools, for example, promise two basic advantages over traditional methods of programming—a shorter, more flexible development cycle and ease of use, whereby even a 'fair-weather-only' programmer would be able to build reasonably sophisticated applications.

RAD tools take advantage of the fact that prototypes are the ultimate design tool because they ensure that an application's capabilities match a user's expectations. RAD makes it easy to develop 'mock-ups' of an end-user application, much faster than would be possible with traditional programming tools. This enables developers to show a prototype application to the end user.

RAD tools make it easy for developers to build application interfaces by assembling buttons, menus, windows, toolbars and so on, from a pre-existing palette of components. After placing all the necessary components on the application window, the (non)programmer can simply hook up the visual components to related commands—simply by placing these 'programming constructs' on a form, and describing their behaviour through the use of an associated scripting language (without having to write a single line of code). This 'model' is shown to the end user, and once a model is agreed upon, the developer simply creates an application that looks and behaves like the prototype.

A second, lesser known, category of visual programming languages is one that follows the visual modelling approach. These can be used to create 3D visual models of systems and programs, and then simply 'execute' the model to simulate its actual operation. Physically, of course, these models do not have anything in common with an actual system, but they can accurately represent flow of data, and help design an application that can control it, in a very intuitive way.

Code as a commodity

So all those wizzo-words—OOP, Visual Programming, RAD—sound good, right? But wait—there is a flip side.

l 1972 - Dennis Ritchie invents the 'C' programming language.

l 1975 - Bill Gates and Paul Allen write a version of BASIC, which they sell to MITS (Micro Instrumentation and Telemetry Systems), who produce the Altair, an 8080-based microcomputer.

l 1976 - Design System Language, considered by some to be the forerunner of PostScript, makes its appearance.

l 1980 - Bjarne Stroustrup develops a set of languages referred to as 'C with Classes'—these later form the basis for the C++ language.

l 1982 - PostScript arrives on the scene. It is, essentially, a 'page-description language' which is optimised for graphics and text, and paves the way for the desktop-publishing revolution of the late-1980s.

l 1983 - Microsoft and Digital Research release the first C compilers for microcomputers. In November the same year, Borland releases its Turbo Pascal.

l 1986 - C++ appears. The name is coined by Rick Mascitti.

l 1989 - C++ 2.0 arrives, and C++ 2.1 makes its appearance in 1990, and for the first time, includes exception-handling features.

l 1991 - Object Oriented Programming (OOP) gets serious. Visual Basic arrives on the computing scene, and establishes reasonable presence in a short time. Microsoft incorporates Visual Basic for applications in Excel.

l1997 - Release of Visual Studio 97 from Microsoft.

PROGRAMMING FOR THE FEATHERWEIGHT CLASS

For most of us, our Pentium PCs, with MBs of RAM and GBs of disk storage space, constitute our computing universe. But another world exists—a world with a computing realitydifferent from ours. In this world, programmers have to design applications for 8-bit processors such as the Zilog Z8, and the Motorola HC05.

These IBPs (itty-bitty processors!!) run at clock-speeds that are less than one-tenth that of our PCs, and are often paired with as little as 16 KB of RAM. IBPs are, usually, single-chip devices where the CPU, RAM and ROM must all coexist,

and are most often used in devices which interface directly with the real world—cellular communications equipment, medical-imaging equipment, electronic toys (Furby and Barney,for example), and of course, PDAs.

The processor is small, and the amount of memory available for processing is very less. To compound the situation, the device in question must often respond in real-time (or in near real-time) with 100 percent reliability at all times. This presents programmers with an environment in which programming routines must fit in tens of bytes, and must be optimised for speed as well as space. 'Programming' assumes a different dimension altogether.

Small is elegant

So how different is this reality? Consider the example of Chip Gracey. A software engineer with Parallax, Gracey wrote the on-chip code for the company's elegantly designed PIC 16C56 development system, which consists of two ICs, one 4 MHz oscillator circuit, and a voltage regulator, all powered by a 9V transistor battery. Gracey had to program in BASIC, and had all of 1 KB of code space available for his program. When the program was finally up and running, there was one last bug to be fixed. Gracey took four days to find and fix the bug—not because the bug was hard to find, but because it entailed the adding of an instruction to the code. And why, you might ask, should that be difficult?

Simply because there was no space. Gracey ultimately had to figure out how to make one routine shorter by an instruction, and with incessant looping and code-interdependence (changes to any instruction sequence affect the performance of several routines), that was no easy task.

The point is, Object Oriented Programming and pre-structured, reusable code might work just fine for developing bloated applications for Pentium-III PCs, but the approach does not work for IBPs.

An off-the-shelf generic routine will not make use of memory-conserving and/or performance-boosting features which may be specific to any one processor, therefore the programmer must become an algorithm innovator, and write processor-specific code, for maximising operating efficiency.

For hardcore programmers, it should be reassuring to know that assembly language programming is alive and well, at least in the world of small processors.

Visual programming makes people think they can start programming right away. This is true only to some extent. Many programs require custom-designed functionality, and for the programmer who cannot actually program, that presents a difficulty. Object Oriented Programming languages force a programmer to think from the user-interface downwards. Programmers learn to think of one screen at a time, which can distort their perspective of the big picture.

RAD can be as bad. The iterative design process that RAD tools employ simplifies programming, but the applications designed using RAD usually require a lot of reworking.Multiple redeployment cycles waste much time, and the resulting 'prototyping death spiral' becomes a process where the programmer gets it wrong many times before he gets it right.

The Prototypes created using RAD tools rarely translate into final, working applications, and when they do, they often turn out to be slow. This is because most RAD tools use interpreters, and interpreters usually execute much slower than most compiled code. Such applications are also difficult to port to different platforms, because most RAD tools are platform-specific (Microsoft Visual Basic supports only Windows apps, for example). To overcome these difficulties, programmers might actually build the actual application using traditional languages like C, once the RAD prototype is approved, which means that the application is actually built twice—a colossal wastage of man-hours!

Code is a common commodity in the world of RAD/OOP, and reusability is a prime concern. Although code reuse can be a good idea (it lets programmers modify an application in just one 'location', and the changes would be reflected throughout—pop-up windows, data windows, dialog boxes, and other user-interface elements), it does not happen on its own. Developers need to put in time and effort to maximise code reuse, and often, short development times do not permit them to do so. Also, piecing together pre-written packets of code is not as efficient as writing routines and sub-routines manually.

Back to the future?

So where do we go from here? Is it time to go back to traditional methods of programming? Should we abandon RAD? No, there is no going back. New programming technologies have made a significant impact upon applications development. And programmers must take the rough with the smooth. What has, perhaps, been ignored is that technology cannot serve as a substitute for programming skill. Newfangled programming tools simply bypass the need for good programming skills, and soon, we will have a whole new generation of 'programmers', who are essentially, non-programmers! That, ultimately, is the paradox we face.

SAMEER KUMAR