Mass Market Computers for Software Development

W. Morven Gentleman and Marceli Wein
Software Engineering Laboratory, Institute for Information Technology
National Research Council of Canada
Ottawa Ontario Canada K1A 0R6
<surname>@iit.nrc.ca

Abstract

The mass market or appliance computer has become a plausible alternative to workstations for software development. While it is easy to use for software development, a different style of development emerges. By being truly personal, the mass market computer provides a focus for all the professional activities of a developer. The primary form of software used on such computers is shrink-wrapped software; this has profound implications on how the software is used and the make-or-buy trade-off takes on an entirely new meaning. This paper describes the advantages of mass market computers. As an example, the paper presents how an environment, consisting of a network of mass market computers, has influenced the development of a novel source management scheme.

Introduction

This paper addresses issues related to using mass market or commodity computers for software development. Three principal arguments are presented here. First, software development involves two complementary activities: actual software development and other professional activities, such as preparation of documentation and of reports. Second, there are significant advantages and increased convenience if the two activities are supported by the same computer. Third, the computer that is used for both activities must be truly personal, as personal as one's desk and chair. Furthermore, these activities are best supported with as much off-the-shelf software as possible. Simply put, a large installed base of similarly configured machines, which offer binary compatibility to developers, has resulted in economical yet powerful shrink-wrapped software. As well, the cost advantage of having truly personal computers supports the principal thesis that mass market computers offer an excellent base for software professionals.

This paper presents, as an example, the use of mass market computers as serious tools for software engineering in the Institute for Information Technology of the National Research Council of Canada. The style of software development that was adopted for the environment of a network of personal computers is reviewed. The use of these computers is based on a consistent scheme for software engineering that assumes that there is no central host or file repository.

A more detailed presentation of the software development methodology was given elsewhere [1]. Some of the rationale for the approach and a discussion of its merits are repeated here.

Mass Market Computers

The definition of a "mass market computer" is presented here in order to reinforce the awareness of its potential. These computers tend to be dismissed by software professionals and to be relegated to the status of tools for support staff.

Mass market computers are those that have shipped in very large numbers and, consequently, created a large third party market. This discussion is restricted to microcomputers that are sufficiently powerful to offer a development environment comparable to mainframe, Unix timesharing, or Unix workstation environments. This means, effectively, (a) Apple Macintosh, preferably from the Macintosh II family, (b) IBM PC/AT and clones (286, 386, and 486) or (c) IBM PS/2 (386 and 486). For this purpose, these machines would be configured with at least 4 megabytes of RAM and at least 80 megabytes of internal disk. It is assumed that the computers are networked. In cases of (b) and (c) additional boards are required to provide a network interface (ARCnet, Token ring or Ethernet). Hobbyist machines (Atari 1040, Amiga 500, etc.) have low hardware prices and these have led their purchasers to the expectation of low software prices. As a consequence, these machines are unattractive for serious commercial software vendors, and hence the range of third party products is limited.

Operating Systems

Mass market computers can be used with various operating systems, including several flavours of Unix. However, the native operating systems have advantages that are an important part of why these machines are a plausible option for a development environment. Consequently only MS-DOS, OS/2, and the Macintosh OS will be considered. Also, these machines can be inexpensively configured to provide a rich human interface with graphics and sound, to provide substantial local RAM and disk, and to be networked for communication between each other, with shared resources, and to remote computers. It is important that the minimal configuration have these qualities, so that software can count on these base features, and need not allow for a lower common denominator.

How does this environment differ from mainframe or Unix development environments? The difference is primarily attitude, not functionality. The functionality of OS/2, for instance, is essentially the same as that of Unix, with the additional features of multiple execution threads per process, high performance IPC, and dynamic linking. The PS/2 hardware functionality is comparable to a typical Unix workstation: a reasonably powerful processor, adequate RAM and local disk, pointing device, good graphics and sound. The

difference in attitude is that the mass market computer is viewed as an appliance, rather than as a machine to program. This difference in attitude is partly on the part of individual users, but primarily on the part of the marketplace as a whole. That is, individual users must understand and deliberately exploit the ability to personalize the environment, but this is only possible because the manufacturer, third party suppliers, and the great majority of the purchasers treat these machines as platforms for off-the-shelf software.

The Computer as an Appliance

The concept of a personal computer as an appliance is loosely credited to Steven Jobs as a way the Apple Macintosh was marketed. The phrase did not refer to a box dedicated to a single function, but rather to the user's expectations of a large marketplace of shrink-wrapped software. The key issue in perceiving the computer as an appliance is that programming is expensive and should be done only where the cost can be amortised across many usages. Consequently, a scientist or an engineer, who has spent many years programming, and even the professional programmer, who earns his living programming, are better off using a professionally polished commercial product than hacking up something themselves. We accept that the average physician or accountant is not a skilled programmer and can buy a program to do a given task more cheaply and efficiently than if he or she wrote the program. The same rationale now also applies to software professionals. The essential point is not that the user could not produce a good program, but that, unless it is worth focusing effort on producing a competitive one, the investment in cost and time is ill spent. Hence, the main form of software used on appliance computers is shrink-wrapped software. Thus, the software engineer using an appliance computer is likely to be both a consumer of shrink-wrapped software and a producer of such software. It is interesting to note that this is the natural extension of the Unix philosophy of having tools and combining them instead of programming — the extension being that, in general, tools should be bought, not built. As with Unix tools, uniformity of interface and integration of environment are essential.

Support for software development implies supporting more than just program development — all the standard program development tools, from compilers to performance profilers, are available on microcomputers for use with programs developed to run on those microcomputers. The performance of these tools on microcomputers is quite comparable with the performance of equivalent tools on mainframe, minicomputer or workstation environments. The software development approaches described later in this paper are directed at the cross-development of software to run on another computer. In the Software Engineering Laboratory, the cross-development is of software to run on industrial multiprocessors. However, appliance computers can and are being used for development of mainframe software.

The appliance computer lends itself very well to supporting professional activities. Productivity aids in all areas can improve performance as well as make the work environment more attractive. The software engineer must document software, prepare design and other documents, prepare presentations and carry out project management activities. Personal databases, from bibliographies to calendars, are one place that commercial products on microcomputers significantly benefit individuals. In the case of databases, the simpler ones available on appliance computers are much more suitable for personal productivity than the powerful ones on larger computers. First, they offer better blending with document preparation software and, second, these databases tend to support free-form information, including drawings and scanned-in pictures as data values, and have irregular structure. The formal relational database is not usually what is needed.

A recent, but very useful, trend is to use tools for making presentations. These include the esoteric ones, e.g., projecting live microcomputer display onto a screen, and the more conventional slide-making, editing, etc. Giving effective presentations is an essential part of the life of any professional, and these tools not only make preparation easier, but the resulting presentations are more effective.

Another emerging technology for appliance computers has become known as groupware, which is support for computer-supported collaborative work. This technology offers most of the advantages but not the disadvantages of the old time-shared systems. Among the facilities are tools to support joint blackboards and communal comments on posted documents. Computer mediated conferences were an early effort in this vein, but some of the current ideas seem much more impressive.

There are two underlying network models that tend to be used in networking personal computers. The more popular one is little more than a communications network to access a central shared file server. This structure mimics the old time-shared computers and inherits many performance limitations. The network underlying the approach in this Laboratory is a peer-to-peer distributed network in which each machine is both a server and a client. At any time any one machine can be considered as the central server of some particular database.

Single-User Environment

Most microcomputers are set up as single-user environments, truly personal machines, and this is key to their effectiveness. A single-user environment does not imply running one program at a time. In fact multiprogramming is essential for most users to be able to respond to the interruptions in their work without losing what they were doing at the time of the interrupt. Window-oriented interfaces provide smooth transition of attention between different programs. Also, single-user environment does not mean isolation. The benefits of sharing are accomplished through networking.

One aspect of the single-user environment is that if there is never another user on a given machine, then its performance is predictable, and the user can plan his activities knowing how fast or slow his machine will be to help him. For many purposes, having the whole power of the machine dedicated to a single-user means there is power to provide a better user interface. Another aspect of the single-user environment is that the user can set the machine up to his own preferences. Admittedly, time-sharing systems often had a user profile that facilitated some personalization, but rarely did it go as far as the individual setting of key repeat rates, mouse speed multipliers, window placement and default open windows, and so forth that typifies personal computers. One particularly important personalization is the software release being run: being able to run an old release can mean old programs can be run again, not moving onto a new release immediately can provide stability while meeting some urgent deadline, going to a new release early can provide access to some new service not of interest to the community as a whole.

The fact that a single-user machine provides better crash isolation is a mixed blessing. Many single-user systems are unreliable and

crash more often. A crashed single-user system may have fewer tools for investigating the crash. On the other hand, the user is less likely to be affected by crashes caused by other people and may be willing to try things that would be regarded as discourteous on shared systems exactly because they might crash. Of course, a network-related crash is not necessarily isolated.

Levels of Interaction with the Remote World

Interacting with remote computers, especially mainframes running preexisting programs, is commonplace for mass market microcomputers and is well supported. It is essential if program development for mainframes is to be done on the microcomputers. The first tool is the terminal emulator. Typical terminal emulators have a number of features that make them far more attractive than the terminals they replace.

The first feature is the existence of macros and an autopilot mode. In striking contrast to the assumptions behind X windows, the common assumption in the microcomputer world has been that the program running on the remote, and its interface to a terminal, are given and immutable. The user at his microcomputer, however, might prefer a very different interface — and if he can program his microcomputer through a communications control language in his terminal emulator, there is no reason why the remote program output should be what he should see on his screen, nor is there any reason why input formats required by the remote program should be what he must enter. Such a locally generated interface can even hide the fact that logon to a remote computer took place!

Let us give an explicit example of the benefits of distinguishing between the user interface and the interface to the remote program. There is a suite of programs developed in the 1970's for analyzing vehicle dynamics of trucks, and as is typical of programs of that era, the input data set describing a specific problem to be solved is a deck of cards, with a complex format of numeric values in fixed fields. These input data are tedious and error prone to prepare. A graphics interface, where one can point to locations on an drawing of a truck and enter the equations and parameters describing behaviour of that part, such as shock absorbers or anti-lock braking devices, is a much better way to define problems, and actually operates by building the corresponding "card deck" and submitting it to the analysis suite running on a remote computer.

Another feature found in terminal emulators is the ability to run multiple sessions, possibly even to different computers. This ability often provides services not directly available within a single session, especially when the terminal emulator supports the ability to select something on the screen of one session and send it as if it were keyboard input to that or another session.

For many systems, such as Unix, a scrolling tty is a common idiom for a terminal. The feature of a large history buffer, with scroll bars, multiple panes, and the ability to make selections to print or save in disk files, makes the emulator far more attractive than the real terminal it replaces.

Also, the ability to drop into a wysiwyg editor to form text that can be sent to the remote computer expecting input, regardless of what editing facilities the remote computer would itself provide at that point, is highly attractive.

Software Development

First, let us present another argument as to why using a multiplicity of computers is a good basis for software development. It is a common observation that, despite the best of intentions, users of any single system tend to build dependencies on it into their programs. Those not deliberately motivated to ensure portability are particularly susceptible, either out of ignorance (it never occurred to them things might be different) or out of laziness (it is easiest to do what the local gurus tell you). The fact is that, if the work of someone on one microcomputer is used by others on different microcomputers, community pressure improves the portability of the resulting program. For example, software actively used on many personal computers exhibits less dependence on unique local aspects, e.g., file locations, configurations of hardware, country/human language dependencies, etc.

In general, software development on microcomputers has taken an evolutionary path from former time-sharing systems through the creation of host-centred networks. Former host-based tools have continued to be used. In particular, source management schemes, such as SCCS [2] and RCS [3] are still employed, which imposes the requirement for central repositories of files and mandates the mechanism where a file to be changed must be signed out from the central repository.

The software development system in the Software Engineering Laboratory at the National Research Council takes a radical departure towards a distributed system, including provisions for supporting intermittently connected developers. It is being used in the development of the realtime multiprocessor operating system Harmony [4]. The scheme has been in use since 1984 in the maintenance and evolution of the Harmony operating system itself, for the maintenance and evolution of various application programs intended to run under the Harmony operating system, and for the maintenance and evolution of various tools employed to develop Harmony programs (these tools must be portable across many operating systems from VMS to the Macintosh OS). The scheme has also been used for assorted unrelated programs written by people familiar with this approach. The scheme (called DaSC for reasons given later) is described in detail in Gentleman et al. [1] and hence only a brief description is given here. It is presented because the approach was influenced heavily by the fact that it is being used on a network of mass market computers.

The source code and configuration management scheme is based on the concept of overlapping layers. This concept draws on the analogy to the film animator's cel (a sheet of celluloid) as a way of thinking of, and representing, change. By overlaying a transparent medium on an existing image, the animator can paint on the medium to add to the image or to paint out parts of the image, giving a new composed image without violating the integrity of the original image. The painted layer of transparent medium concisely defines the changes made in the composed image. It provides an identity for a set of changes that is retained, so before and after images are readily available. Disjoint changes produced by separate animators can readily be verified by comparing their layers, without resorting to the original image, and can be applied in any order. The cel can be used to support temporary experimentation (like delayed posting in database systems). Obviously the original image might itself be composed of many such layers. It is evident that animation cels also support reusability. The analogy to the animator' cel led us to call our system Database and Selectors Cel —DaSC.

The source management scheme depends on how the source code is organized into files. The entire set of functions making up the program is stored in a file tree where each function or a set of declarations is stored as a separate file; multiple functions are never placed in the same file. Each layer contains a portion of the files. The entire system of files consists of several overlapping layers. Instead of having a single, central master copy of the file tree, each developer has an identical read-only copy of the entire file tree in the bottom layer of the multi-layer view. Each developer has additional private layers that contain work in progress as various modified functions. Each function that has been updated overlays the original one that exists in a lower layer. The most recent version of the entire program, e.g., operating system kernel, consists of all the required functions, where each function is picked up from the highest layer in which it exists. The scheme uses little network bandwidth during development because almost all accesses are to the local set of layers, including the read-only master layer. If necessary, the developer can even work entirely detached from the network.

This style of configuration management is "free-wheeling," because the developer works on a set of functions in his own environment, without checking them out from some central depository. Instead, the consistency of the whole is solved at the stage when the working layers of different developers must be consolidated into a new master version which will then be distributed as their new read-only Master.

There is a well-established body of literature supporting this "free-wheeling" approach. Our approach draws on the experience in concurrency control in accessing and updating B-trees which has been a subject of intense study [5–7] and more specifically on the optimistic concurrency control introduced by Kung and Robinson [5]. The simple observation motivating the approach of Kung and Robinson is that in the case of infrequent collisions during simultaneous access, improved performance is achieved if multiple threads make a modification assuming no collision, but are prepared to back out if a collision is detected. The approach assumes that there is a scheme for detecting collisions. In the case of updating B-trees the optimistic concurrency control has offered many advantages over the process of locking all access to portions of the tree during update.

The scheme in DaSC also takes the optimistic approach that the management of people is such that they work in different areas of the system (don't assign two programmers to work on the same feature) so as to keep collisions to a minimum. The tools for merging layers into new composite layers are automatic: for each function the version in the highest layer is found. However, collisions require human intervention. That means that when a layer is consolidated, everyone else must check that the layer being consolidated does not interact with the layer he is working on, or correct the work in his layer to accommodate for any interactions that are found. Our experience is that such rework is minimal in practice. Thus, regression testing of the entire new system is done before the composite working layer is consolidated into the new master layer. Distribution of a software release to existing users can be by shipping selected layers only. They can apply it to their current release after resolving any interactions with work they have done since the common base release, which will be represented by the layers through which they see the common base.

The actual mechanism for retrieving individual functions uses the inclusion mechanism of the C compiler preprocessor. For example, a fragment of an inclusion file shown below:

```
#include "Master/harmony/bound/src/getword.c"
#include "Working/harmony/bound/src/lookahead.c"
```

selects the function getword.c from the Master layer and the function lookahead.c from the Working layer. As stated above, each #include statement points to a file that contains only one function. It is self-evident that an application consisting of several hundred functions will be defined by an inclusion file with several hundred inclusion statements. On the other hand, there are no inclusion statements in any of the functions.

The Software Engineering Laboratory has used DaSC on projects involving of the order of 10 programmers, some of whom worked at remote locations with intermittent network connection. The applicability of our source management scheme to larger projects is based on the observation that DaSC has much in common with the process of distribution of system software by vendors to a large base of mainframe installations. Each new system arrives as the new base layer to which all local enhancements that were done to the previous system must be ported. Usually, these local libraries are kept in shadow libraries.

One other aspect of source code preparation and documentation is worth pointing out. Among the mass market computers, the Apple Macintosh has a unified imaging model for text and graphics. It is possible to prepare a function written in C and documented with diagrams as shown in Fig. 1 and to compile it without change by any one of the C compilers. The actual code can be enhanced with type styles for readability, a scheme that has been demonstrated in [8]. Among the many wysiwyg text editors for the Macintosh the Nisus word processor from Paragon Software keeps the formatting information hidden from any process that reads the file. As a consequence, the Nisus document shown in Fig. 1 is completely acceptable to a compiler without any filters for removing the style and graphics information.

Mass Market Computers — Advantages

The computer is personal

A mass market computer can be considered as a truly personal workstation and can act as the focus of all professional activities. While in some research laboratories it is accepted that a $30 to $40K workstation represents a reasonable personal tool, in fact in many organizations such workstations tend to be shared, thus destroying the principal advantage offered by a personal computer.

Lots of third party software

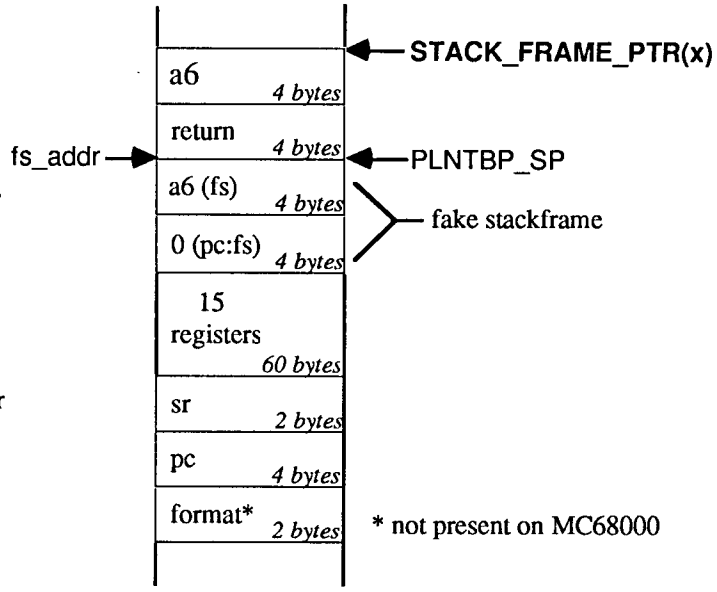
The software for personal computers is well served by the third party marketplace. There is a wide range of useful software. It is critical that software products must be available when needed. It is too late to start when they are required. Planning a useful configuration thus involves choosing what products from various vendors to integrate. It is also critical that the software be reasonably priced, otherwise maintaining a wide selection is infeasible. Economy of scale is essential to the price being reasonable. Customization is not practical beyond the tunability built into commercial products

trace.c

```
void _Trace( x )
        uint_32 x;
/*
```

The second level handler for trace exceptions on a Motorola MC680x0.

The TRACE exception signals that the execution of a breakpointed instruction has been completed and the breakpoint should be placed back at the breakpoint address. The address of the unused argument x is used to obtain a handle on the stack. The interesting part of the stack is shown in the figure.



We must patch the status register value to turn off the trace bit and to set the interrupt mask correctly. Also, we must replace the instruction at the breakpoint address with the one that was saved by the breakpoint trap exception handler before the original instruction was placed there for execution (normally, this is the breakpoint instruction, i.e. TRAP 0).

Note that interrupts have been disabled since before the breakpoint trap exception handler set up the instruction emulation.

Upon taking an unexpected trace exception, we send a message to **_Gossip.**

```
*/
  {
    extern   uint_16     *_Dbg_addr;
    extern   uint_16     _Dbg_instr;
    extern   uint_16     _Dbg_sr;
    extern   uint_32     *_Loc_gossip_id;
    extern   void        _Clr_i_cache_entry();

    register uint_16     *sr_ptr;
    uint_32              fs_addr;
    struct STD_RQST      request;
    struct STD_RPLY      reply;

    if( _Dbg_addr )
      {
        fs_addr = *(uint_32 *)STACK_FRAME_PTR( x );     /* location of fake stackframe */
        sr_ptr = (uint_16 *)(fs_addr + 68);             /* location of SR on stack */
        *sr_ptr = (*sr_ptr & 0x00FF) | (_Dbg_sr & 0xFF00);
        _Clr_i_cache_entry( _Dbg_addr );
        *_Dbg_addr = _Dbg_instr;
        _Dbg_addr = 0;
      }
    else
      {
        request.MSG_SIZE = sizeof( request );
        request.MSG_TYPE = TRACE;
        reply.MSG_SIZE = sizeof( reply );
        _Send( (char *)&request, (char*)&reply, *_Loc_gossip_id );
      }
  }
```


Fig. 1. This function, written in Nisus© word processor, can be compiled without any modification

— the fact that these products are from third party suppliers is essential. No company has a monopoly on good ideas, and the existence of many companies producing competing products stimulates new approaches. Commercial competition plays an important part in constantly improving the quality, in addition to constantly refining the definition of what products in a particular area should do. Reviews of products, often comparative reviews of competitive products, published in commercial magazines provide an important rating system by which products are judged.

To underline this point, we note that this is where the action is: because of the size of the market, because entry cost is so low, and because of a history of market acceptance of novelty, today most interesting new developments in software first occur in commercial mass market software.

Graphical User Interfaces

As was described earlier, the developer of Harmony-related software works with a large number of files which are distributed in many subdirectories. Harmony comprises close to 4000 files in 600 directories. Consequently, the programmer needs good visual contact with a large portion of the file tree and must have a rich multiwindow editor so as to have easy access to many functions (files) while writing or modifying any one of them. A good graphical user interface helps to give the programmer a rich presentation of the context in which he is working [9].

One of the real strengths of mass market microcomputers is that not only has the availability of a rich man-machine interface become standard, but its use pervades the system. Documents, of course, contain pictures and line art as well as using multiple fonts and point sizes for readability and emphasis. Databases accept graphic objects as field values. Electronic mail systems can paperclip circuit diagrams or digitized sound to mail messages, and the recipient can be expected to have programs to use them. OCR or speech recognition can input values into any program at a point where that program is expecting keyboard input. Speech synthesis allows commentary on a screen display without consuming critical screen real estate or disturbing the display itself. Colour and graphics arts techniques effectively indicate attributes of displayed objects. Appropriate graphic displays are used with the mouse or another pointing device to control computation much more intuitively than could text directives.

As a simple example, an endless problem in symbolic algebra systems has been how to represent the richness of mathematical notation within the limitations of ASCII. Using I as the symbol for imaginary numbers not only collides with its possible use as a counter variable in a program, but also with its possible use to represent the identity matrix. Mapping to unique codings, such as by prefix strings of underscore and similar characters, solves parsing problems for programs but is difficult for people to work with. If, however, the ability to represent and enter a richer notation can be assumed, the problem is alleviated. Even just styles such as italics and bold are an enormous advance, and Greek letters together with other symbols of a mathematical font essentially finesse the problems. Selective on-screen palettes balance memorization of key equivalents against use of limited screen space when facilitating rich input.

Cost and functionality of remotes

Often circumstances prevent some user from benefiting from wide bandwidth local links. It may be necessary to work at home, or on site in some customer's premises, or in some branch office. Computing support is especially important in these circumstances to make up for the other support amenities that are also inhibited. The personal computer can be sufficiently self-contained, and is sufficiently low cost, that it is practical to replicate the normal working environment at such a remote site. (It is also sufficiently portable to make it practical to deliver to such a remote site, even for short periods of time.) This is in contrast to mainframe-based computing, where the drop in bandwidth creates a definite second-class citizen situation, and to the diskless workstation-based computing, where a remote station not only has higher incremental expense, but may imply space requirements for a file server.

The principal deprivation is that file transfer speeds drop, but experience shows that the shortcoming can be accommodated even for weeks or months. A lack of network bandwidth is compensated by having adequate local disk to cache things of interest and perspicacity in choosing what to keep locally. For program developers, it also implies a configuration management strategy that is consistent with remote developers.

Cost of experimentation

So many novel products are appearing that to appreciate them it is often desirable to obtain a copy for evaluation and try an actual application. The low cost means this is the responsible way to exploit the market. Published reviews, and surveys of market penetration, help customers choose products and plan cutovers. They also help developers plan responses to features in competitive products.

Quality of code and maintenance

Perhaps the most satisfying aspect to the customer is that the polish of shrink-wrapped software is far higher than that of software on mainframes, departmental minicomputers, or workstations, whether from the equipment supplier or third parties — the supplier of shrink-wrapped software cannot afford anything less. For almost any product of interest, there is a choice and the alternates are qualitatively different. Sometimes they are complementary, and the serious user wants more than one. Often it is feasible for an organization to let users select their preference, rather than forcing standardization on one. In all cases, however, competitive pressure produces continual and significant advances in offered alternatives.There are no examples where products of a few years ago would be competitive today.

What are the Weaknesses

The appliance computer market has evolved from very basic machines bought by individuals who kept the responsibility for their upkeep. The computers penetrated corporations, bypassing the traditional centres of responsibility for computing. All the mechanisms for purchase, licensing, upgrading, etc. continue to be geared to dealing with individual owners. The result is that in a large organization, where the responsibility has been taken over (grudgingly) by some central services, system maintenance is labour intensive, time consuming, and difficult to manage. Software publishers (but not all) have at least abandoned various schemes for software protection but have been slow to introduce corporate licensing and upgrading. Imagine the reaction of a system manager upon receipt of a couple of hundred upgrade notices for a popular software package. Management of serial numbers alone is an onerous task.

Conclusion

This paper has reviewed some of the qualities of mass market computers and discussed their suitability for being the primary tool for software development. A mass market machine is inexpensive enough to being truly personal and therefore being the centre of all professional activities. It benefits from the availability of a large selection of shrink-wrapped software from many creative developers. As an example, the paper described a scheme for configuration management that is suited to a network of personal computers in which does there is no central file server.

References

[1] Gentleman, W.M., MacKay, S., Stewart, D., and Wein, M., "Commercial realtime software needs different configuration management," *Proceedings of 2nd International Workshop on Software Configuration Management (SCM)*, Princeton, NJ. October 24-27, 1989. Published in Software Eng. Notes, 17(7): 152–161; 1989.

[2] Marc J. Rochkind, "The Source Code Control System,"*IEEE Transactions on Software Engineering*, Vol. SE-1, No. 4, December 1975, pp 364–370.

[3] Walter F. Tichy, "RCS — A System for Version Control," *Software — Practice and Experience*, Vol. 15, No. 7, July 1985, pp. 637–654.

[4] W.M. Gentleman, S.A. MacKay, D.A. Stewart, and M. Wein, *"Using the Harmony Operating System, Release 3.0,"* NRC/ERA-377, National Research Council of Canada, Ottawa, Ont., February 1989.

[5] H.T. Kung and J.T. Robinson, "On Optimistic Concurrency Control," *ACM Transactions on Database Systems*, Vol. 6, No. 2, June 1981, pp. 213–226.

[6] L. Lamport, "Concurrent Reading and Writing," *Communications of ACM*, Vol. 20, No. 11, November 1977, pp. 806–811.

[7] M. L. Kersten and H. Tebra, "Application of an Optimistic Concurrency Control Method," *Software Practice and Experience*, Vol. 14, No. 2, February 1984, pp. 153–168.

[8] R.M. Baecker and A. Marcus, *Human Factors and Typography for More Readable Programs*, ACM Press, Addison Wesley 1990.

[9] W.B. Cowan and M. Wein, "State versus History in User Interfaces," *Proceedings of the IFIPS TC-19 Third International Conference on HCI, Interact '90*, Cambridge, U.K. August 27–31 1990, North Holland 1990, pp. 555–560.

Much of the information on the facilities, tools and evolution of mass market computers exists in popular magazines, such as Byte, MacUser, MacWorld, PC Magazine, Etc., rather than in the traditional archival journals. While no specific references are given here, an interested reader might wish to peruse these publications.

Appendix

The system in our Software Engineering Laboratory is implemented on a network of Macintosh computers. A typical configuration is shown below. There are many equivalent software products. The ones that were selected were the most suitable available at the time of selection. Because the relative ranking changes with time and because the range of choice is wide, no brand names are given below.

Typical hardware configuration
- Mac II(si, ci, fx) with 2.5, 4, or 5 Megabytes — lower models possible
  - 60+ Megabytes local disk, full page display, network
  - nearby laserprinter, 300 dpi scanner available

Typical software configuration
- editors
  - program editors
  - document (incl. spelling checker, integrated graphics)
- outline program
  - hypertext, mathematics
- programming
  - native and cross development
  - standard C environment
  - symbolic algebra
- database, spreadsheet, notecards, bibliographic support, statistics
- communications
  terminal emulation and scripting, running remote canned programs, control of user interface
  mail, remote console, file transfer
- utilities
  document compare, font changer, backup, low level filetool, disk repair tools, tree manipulator, file format converter, annotator, post-it notes, etc.
- project planner (CPM)
- drawing
  paint, draw, 3D draw, scan, animate, trace
- sound, speech synthesizer, sound editing
- forms-based file manager
- accounting
- publishing
  desktop page layout, presentation organizer, hypertext interactive presentation