NetVista: Growing an Internet solution for schools

by W. A. Kellogg
J. T. Richards
C. Swart
P. Malkin
M. Laff
V. Hanson
B. Hailpern

NetVista™ is an integrated suite of clients and servers supporting Internet access for students and teachers in kindergarten through 12th-grade schools. Developed by a small team of IBM researchers, NetVista is a prime example of using an object-oriented framework to support user-centered design and to accommodate Internet-paced changes in network infrastructure, functionality, and user expectations. In this paper, we describe salient aspects of NetVista's design and development and its evolution from research project to product. In particular, we discuss the factors supporting a sustained focus on end users over the life of the project, the object-oriented framework underlying NetVista, and the role of this framework in accommodating both evolutionary and radical changes to the design of the user interface and the underlying technical infrastructure.

Most technological advances are (necessarily) carried out by experts in technology and programming. In computer science research, groups often emerge to push particular technologies (for example, speech recognition or video compression) beyond their current limits. While the potential for new applications may be recognized, the development of new technologies often takes place outside real contexts of use, fostering the need to find applications later—a process sometimes referred to as "technology push." The place of "people experts" in this kind of technology-driven development traditionally has been limited to activities such as performing user interface (UI) critiques, running focus groups, or conducting usability evaluations that largely reside outside of the activity of the primary development team.[1]

Today, organizations such as ACM's SIGCHI (Special Interest Group on Computer-Human Interaction) and UPA (Usability Professionals Association) have gained visibility in the technical community. The importance of "human factors" and user-centered design has become part of the mindset of software managers and developers. Human behavior experts are increasingly part of many development teams. Nonetheless, it is still unusual to see the justification for or the development of technology truly driven from the "outside-in," that is, grown *from* a focus on users and their context of use *to* the technology. By "outside-in" we mean development in which technology is treated as a resource to be used along with other resources, such as knowledge of the practices and characteristics of the users or the usage situation, to create a tool or solution. There are complex reasons why this approach to design and development does

not prevail, despite its proven ability to create more satisfactory results.[2,3] With the advent of the Internet, the prospects for user-centered development have in many ways worsened: new technologies are introduced at a rapid rate, the time pressure to create and release applications is intense, and fads and styles in user interface "widgetry" and effects tend to dominate seemingly more plebeian concerns such as usability.

In this paper we describe the development, from initial concept to shipped product, of a suite of server and client applications aimed at providing Internet access for kindergarten through 12th-grade (K–12) schools. The development project was completed in less than three years, in spite of changes in product direction, organizational participants, organizational support (the product was canceled twice), and underlying software platforms. Furthermore, in the course of development, we did not make use of traditional artifacts such as requirements documents and engineering specifications. Given that we developed a successful product, in a short time, using unconventional methods, we believe that an account of our development practices will be of interest to many.

Our experience with NetVista* proves that software solutions can be developed with a firm end-user focus, in an accelerated time frame, without compromising quality or innovation. It demonstrates what can be accomplished by a small team of persons with diverse talents and perspectives when they remain focused on users and what we might call "usability in the large," which includes the entirety of the user's experience with the solution being proffered.[4] We do not think of NetVista as a model for all or even most software development, although there are important paradigms for which it could be a model—for example, much Internet application development. Some projects demand a large team and a relatively formal, shareable process. Our experience suggests, however, that it is easy to underestimate what can be accomplished by a small, focused team.

NetVista had two origins and from neither could the end product be foreseen. The first was an attempt to bring the Internet to K–12 schools (before the World Wide Web). This origin is later discussed in detail; it formed the technical core of the project. The second origin was an attempt to bring together research technology, IBM software, and IBM hardware to provide a unified, very easy-to-use Internet offering. This second origin resulted in the forming of an interdivisional team that built one of the IBM's first "Web-year"-speed[5] products. It was the combination of the core group devoted to a custom solution for K–12 and an interdivisional product team dedicated to turning out a real Internet product in a very short time that led to the success of NetVista.

We want to emphasize that the work we present here constitutes a report of our experience and of work in progress. It is not meant to be, nor should it be construed as, an experimental or scientific study of software development. Rather, it is an attempt to articulate and extract lessons from what was for us a focused, creative, collaborative work effort in a way that might be useful for others engaged in similar pursuits.

The NetVista project

NetVista began as a research project at the IBM Thomas J. Watson Research Center in late 1993, and evolved through several stages until its release in June, 1996, as IBM's K–12 solution for Internet access. The research motivation for the project included the desire to explore the capability of Smalltalk (an object-oriented programming language and development environment) to handle a communication-intensive client/server application and to simplify the complexity of the Internet and its use, which at the time was fairly daunting, particularly for nontechnical users.

The research group already had ties to the K–12 business unit, so from the beginning the project aimed to create simplified access to the Internet for the K–12 environment. The history of the project reflects both the philosophy of the team to "grow" software in its context of use,[6] and the nature of the underlying programming environment, which allowed changes to evolve with the project's changing direction over time.

The first demonstration of what was to become NetVista occurred less than eight weeks after the initial requirements for the project were articulated. A prototype local area network (LAN) -based system with 14 client machines was unveiled early in 1994 at the IBM Schools Executive Conference in Atlanta, Georgia. During one week, team members spoke to dozens of school administrators, sometimes demonstrating the Internet as much as the software. Literally hundreds of conference attendees used the software to get their first taste of the Internet or, for some of the experienced users, to use Telnet[7] to con-

nect to their home machines to check their e-mail. Messages and greetings were sent, and all involved were excited about the Internet and its potential for education.

By mid-1994, "k12.net," as it was then called (after the name of the Internet domain we used for support), was installed in about a dozen beta test sites in North America. Installing and maintaining these sites was the primary way that we grew to understand the environment in which the software would be used. IBM offered "k12.net" to schools as a Limited Availability Services Offering from this time until the product was released in 1996. As was true with the beta sites, the Services Offering sites were an ongoing source of information and inspiration, generating many changes in the underlying software.

In 1995, IBM's new Internet Division was formed and interest developed in offering an Internet solution for home users. The "k12.net" code was ported from Microsoft Windows** 3.1 to Operating System/2* (OS/2*), given a thorough systems test, and released in the fall as beta software. Named "NetComber*," the code was given away at conferences on a compact disk and made available for download via the Internet. Feedback via e-mail messages to the research team from these OS/2 users provided further grist for working out the client user-interface design and functionality, and for anticipating the range of situations to be faced during installation.

In early 1996, NetComber became the basis for NetVista. The client code (having gone through a number of changes, including the incorporation of Web browsing and a central focus on managing and sharing URLs, or uniform resource locators) was ported back to Windows 3.1 and over to Windows 95**. An administration client was created, and the OS/2 servers were rebuilt and extended. The core research team formed a partnership with IBM's K–12 Solutions business unit to release the product in June, 1996. NetVista 1.1 (adding Macintosh** OS as a client platform and Windows NT** as a server platform) was released in November, 1996, and followed by another release in late 1997. The NetVista Web site is currently at http://www.solutions.ibm.com/netvista.

Understanding the K–12 environment. At the beginning of 1994, when Mosaic[8] was in its infancy and Netscape Navigator** had not yet arrived, few schools had Internet access.[9] Those that did typically had a UNIX** shell account through a nearby college or university, or an America Online** account on a single machine. Teachers fortunate enough to have access through one of these mechanisms often shared their single account with a class of students, printing out individual mail messages in order to distribute them to the intended recipients.

We visited a "model technology school" in New York in the fall of 1993 to see the "state of the art" for ourselves. What we found surprised us. In an elementary school with five LAN-connected computers in each classroom (one with Internet access) only a single teacher was regularly using the Internet in her classroom. The school's technology coordinator told us that the teachers were slowly making a cultural shift,[10] based largely on the efforts of the pioneering teacher. This teacher was excited by the projects she was doing with her students and served as an "evangelist" for the Internet. On the wall outside her classroom we saw a huge map of the United States, with yarn stretching from various cities to the edge of the map, where reports on water quality, gathered from the Internet, were posted. This teacher's class was also participating in a "virtual vacations" project, in which students contributed to a database describing what it would be like to vacation where they lived. Students who contributed earned the privilege of "taking" vacations in other locations about which students around the world had written.

Teachers fortunate enough to have access to the Internet often shared their account with students.

Seeing the use of the Internet by the teacher was exciting to us, but the "take-home" lesson was obvious: if her school was the "leading edge," other schools would be even less ready to take advantage of the Internet. We needed to get a better idea of what was really happening, both from a technical infrastructure point of view—what capacity machines and networks were being used—and from a sociological point of view—were teachers ready to adopt this technology? What would help?

We invited a group of educators, who were pioneers in bringing Internet technology into schools, to be-

come an advisory board for our project and to join us for a one-day workshop on requirements for wide-scale Internet access in K–12 schools. We asked these experts to describe the most important requirements from their perspective. What they told us helped us to understand the essential requirements for supporting Internet access in schools, and also helped to determine our priorities. First, they had needed administrative integration with the existing Novell, Inc., servers that ran their schools' LANs (but they did not want the servers modified in any way—the servers were critical to the running of their curriculum courseware; server support remained something of a "black art" to the local administrators). E-mail (electronic mail) and Gopher were their top-priority applications; news, chat, Telnet, and FTP were useful, but less important.[11] "What about the Web?" we asked. Less important than Gopher, they told us—it takes too long to download all the graphics.[12]

The educators also told us that blocking access to inappropriate sites was of paramount importance. Without protection for students, Internet use would never become widespread. They needed software that would run on Macintosh as well as IBM-compatible platforms. They asked for a solution that was easy and cost effective to install and administer. They told us that the common workstation being installed in the classroom (in 1993) was a 4-MB (megabyte) machine, sometimes lacking both a local "floppy" drive and a local hard drive. And they pointed out that these machines were shared rather than dedicated to a single user. Software based on the personal computer model ("my files reside on my hard disk") would not work in schools.

Design objectives. Based on our growing understanding of the K–12 environment, we were able to begin to particularize our objective of creating a simple, sensible Internet experience for users. We knew that the software needed to be easy for users to work with and provide nearly effortless access to Internet content while drawing little attention to itself. Since the same software would be used by students and their teachers (who would generally learn it first) it needed to appeal to children while not seeming childish to adults. It needed to be almost immediately usable, since many users would get no more than one or two hours of Internet time per week and they could scarcely afford to spend that time mastering the software. It needed to present a consistent model of the Internet across a number of different and separately evolving Internet protocols. It needed to shape the behavior of users who were mostly new to the Internet so that they would be welcomed by those already experienced in its use. It needed to be easy to install and maintain on dozens or hundreds of client machines at once, and it had to be easy enough to administer that a teacher could do it on a limited part-time basis.

Our initial focus on the characteristics of the users and their environment enabled us to begin to understand what simplicity would really mean for each user. As our work progressed, we came to see simplicity in terms of three constituent features of design: *usefulness*, *appropriateness*, and *usability*. Usefulness meant that every function we included had to be justified. Function had to be important to most users, most of the time—not just to experienced users, or useful in conceivable but exceptional circumstances. Being included in other software (e.g., popular e-mail programs for personal computers) was not sufficient justification for us to include a function; rather, we sought a minimal set of simple, elegant functions that would allow most users to do most of the things they would need to do most of the time.

Appropriateness meant that even if something was useful, it also had to be needed by K–12 users in schools. Our design choices had to enhance Internet use for teachers and students, and remain compatible with the situation in which they would engage in Internet activities. Issues such as how to store personal information (e.g., bookmarks, nicknames, and signatures) when machines are shared and possibly without hard disks are a case in point. More generally, teachers and students would use the Internet in some ways more than in others (for example, e-mail and the Web more than direct FTP and Telnet), and we felt that our design should reflect those usage biases. Simplicity could be enhanced by matching the functionality we provided to the specialized use that would occur in the K–12 environment.

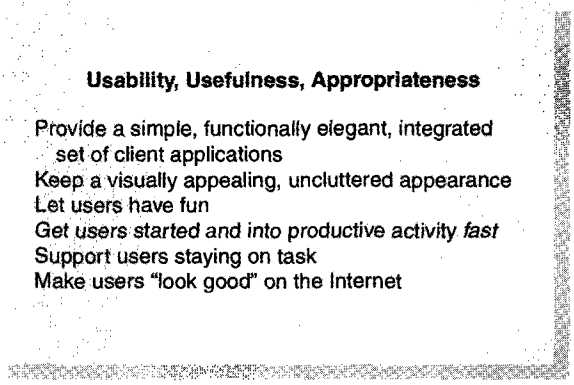
Usability meant that whatever was finally included in the application had to be easy for Internet novices to comprehend and use, but not get in the way of experienced Internet users. So, for example, the client programs (e.g., Gopher, e-mail) had to be easy to use by persons with little knowledge of the Internet or what it was for, and the administration client program (for managing users' mailboxes, newsgroups, and security) had to be understandable by a teacher or school administrator responsible for maintaining the server.

Figure 1 shows some of the more detailed objectives that arose from considerations such as these. Some of these design goals, such as simple, integrated function and an uncluttered visual appearance, were about how the software would look and behave. Others were intended to keep in mind effects that we wanted for our users, such as getting them started quickly and productively on the Internet, and helping them to conform to the interactive conventions of the Internet (i.e., to observe "netiquette").

The team that created the function to be included in NetVista and its user interface worked with a shared background of design concerns and human-computer interface principles. These included consideration of function, its presentation, and interaction with the user from many perspectives, including that of perceptual-motor coordination, the user's conceptual model of the software and the Internet, support for tasks in using the Internet,[13] and support for social practices that arise in any community of Internet users. For example, we knew sharing information with colleagues to be crucial for a new community of Internet users. Principles of human-computer interaction (HCI) have been articulated by many HCI researchers and designers.[14–20] We shared many of the perspectives, concerns, and principles espoused by these authors, and such issues surfaced again and again in the evolution of NetVista. Here we offer a sketch of some of the concerns that drove our design work and how they applied in the design of NetVista.

We addressed interaction and tasks in NetVista at a wide variety of levels. For example, the lowest level involves perceiving the screen and the state of the system. We carefully considered the interface and the movements necessary for carrying out actions at this level. This led us to consider younger children who could not type, as well as those who were experienced touch typists. We minimized the need for reaching for the mouse during heavy text-entry tasks, such as adding several names in a row to the nicknames list. We made it possible to do all common tasks without typing (although the body of text messages had to be typed). We simplified screens to draw the user's attention to the place where the action would begin (e.g., in the Send Mail display, the cursor blinks in the "to:" field in a visually simple, streamlined message header—see the Send Mail window in Figure 2). We also took care to position buttons across screens used for certain high-level tasks (e.g., deleting mail) so that the need for positioning the cursor and moving the hands was minimized.

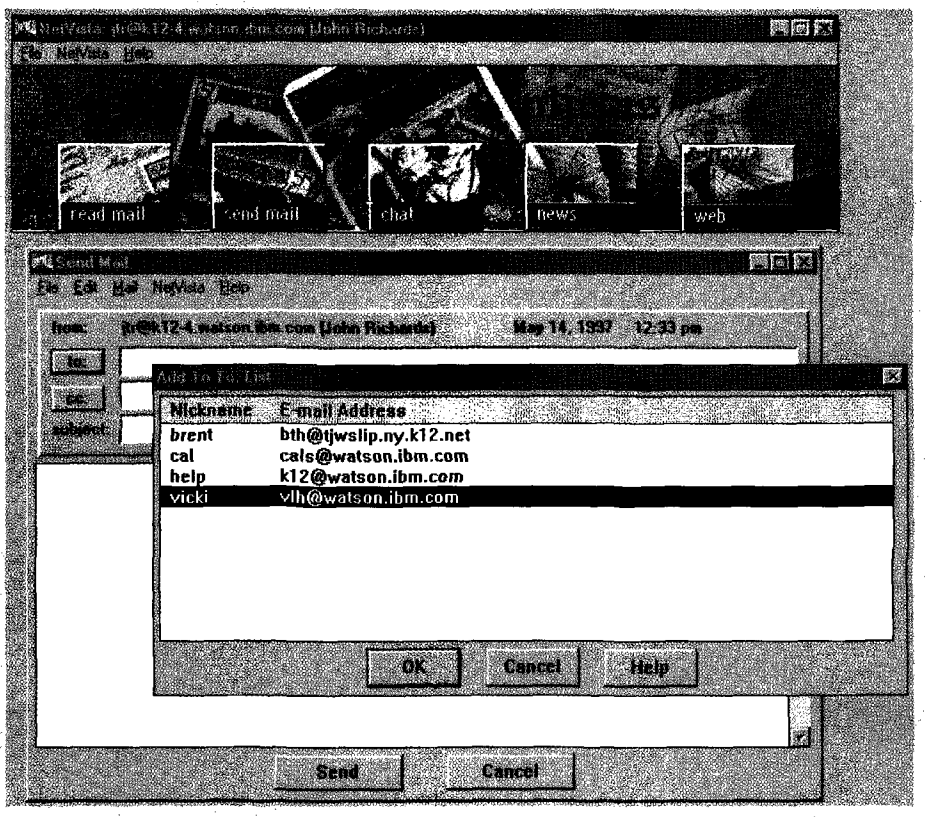
Figure 1 Some of the initial design considerations for NetVista



At the task level, we focused on creating simple "plan-act-evaluate" profiles. The notion of plan-act-evaluate cycles in human-computer interaction comes from Norman.[21] Applied in practice, this principle meant making simple, basic things easy for users to "figure out" and do (and, of course, we wanted to make more advanced, difficult things possible to do). It also meant supporting the evaluation of system state so that the user could easily infer what else might be needed to accomplish a task. Feedback, acknowledgments, and informative messages are essential in supporting the user's ability to evaluate the system state. When users attached a file to an e-mail message, for example, a paper clip appeared over the right top corner of the mail header to indicate attachment. This is a simple response from the system, but it provides immediate visual feedback that an attachment has occurred.

Another application of this principle to the NetVista interface was the general strategy for presenting function on buttons, menus, or pop-up menus. To warrant its own button, a function had to be essential to the window and frequently used—for example, the "Send" button on the screen used to compose mail. Menus contained less frequently needed or integrative functions, such as "Add Sender to Nicknames List." Pop-up menus contained special global integrative functions (such as "send this in a mail message") that could be applied anywhere on the screen (as in bookmarking a Web page, or pointing to a URL or e-mail address in text to use it). When

Figure 2 The NetVista launch window is shown at the top. Clicking on the "send mail" icon yields the (underlying) mail window directly below for composing a message. When the "to:" button is clicked in the mail header, the user's list of nicknames is presented.



this strategy is implemented, users implicitly experience a world in which more important and pivotal functions are more perceptually salient. Advanced function can be encountered and exercised as the user feels ready. Thus, in such a design there is a kind of scaffolding for learning (both intentional and incidental learning) and progressive disclosure of function that is at the user's discretion.[22] Adhering to this principle also contributed to our goal to provide an uncluttered visual appearance.

Finally, another major concern in our design work was providing support for error interpretation and recovery. Of course, wherever a design can preempt an error, it does not need to address how the user will recover from it. The Macintosh (and now widespread) convention of graying out unavailable menu commands is an example of preempting errors. It prevents users from fruitlessly trying to execute function that is unavailable due to system state (see Carroll, Kellogg, and Rosson[23] for a description of Training Wheels, another approach to guiding users by manipulating the availability of function). Graying out unavailable or inappropriate functions can also simplify the user's learning task. For example, in the user's nicknames list, we grayed out the "Send to:" and "CC to:" buttons whenever a host site, rather than an e-mail address, was selected. This helped new

Internet users to grasp the difference between e-mail addresses and host site names.

One weakness of the graying out technique, of course, is that it does not tell the user *why* the function is currently unavailable, or what would have to be the case for the function to become available. We observed one user trying over and over again to open a grayed-out newsgroup because that was the *only* newsgroup she wanted to see (and she wanted to see it very much). She did not notice that the newsgroup was empty (had zero articles), and she either did not notice or did not understand the meaning of the newsgroup being grayed out. Her experience led us to a small innovation in how we handled grayed-out functions: the second time the user clicked on a grayed-out item, pop-up text would explain why the item was unavailable and under what circumstances it would become available. This design is an example of unobtrusive and context-sensitive support for novice users (as contrasted with unobtrusive support for experts, discussed below). It helps the novice user without burdening the expert or the user who knows what grayed-out items mean but has made a clicking error. Thus, the explanation is only invoked when a user clicks *twice* on an unavailable item—an unlikely behavior for an expert. And a user who makes a pointing or clicking error (once) will not trigger the explanation, thus taking into account the perceptual-motor coordination dimension.

Where errors can be made, users need support to recover from them—both to recognize the nature of the problem, and to remedy it in whatever way is most desirable. Whenever possible, NetVista reinstated the context in which an error occurred after offering an explanation of what was amiss. For example, nicknames for people and host sites had to be unique. If a user tried to add a second e-mail address with a nickname that was already in use, for example, NetVista would respond that the nickname was already in use, and then re-present the dialog box where the user had specified the duplicate nickname. The nickname was already highlighted, ready for the user to type a different nickname. This design helps users resolve the problem, in multiple ways: it says plainly what the problem is and what needs to be done to remedy it; it puts users right where they need to be to take the requisite action; it reminds them of what nickname they tried before, and the highlighting allows them to just think of a different nickname and type it (i.e., without having to backspace, or click the mouse, or make any superfluous action). The dialog thus boils down to its essence, with NetVista in effect saying "that nickname is already taken," and the user responding "well how about this, then?" This kind of design detail seems like a small thing, but attended to along with many others, it builds both a solid sense of usability and the user's trust.

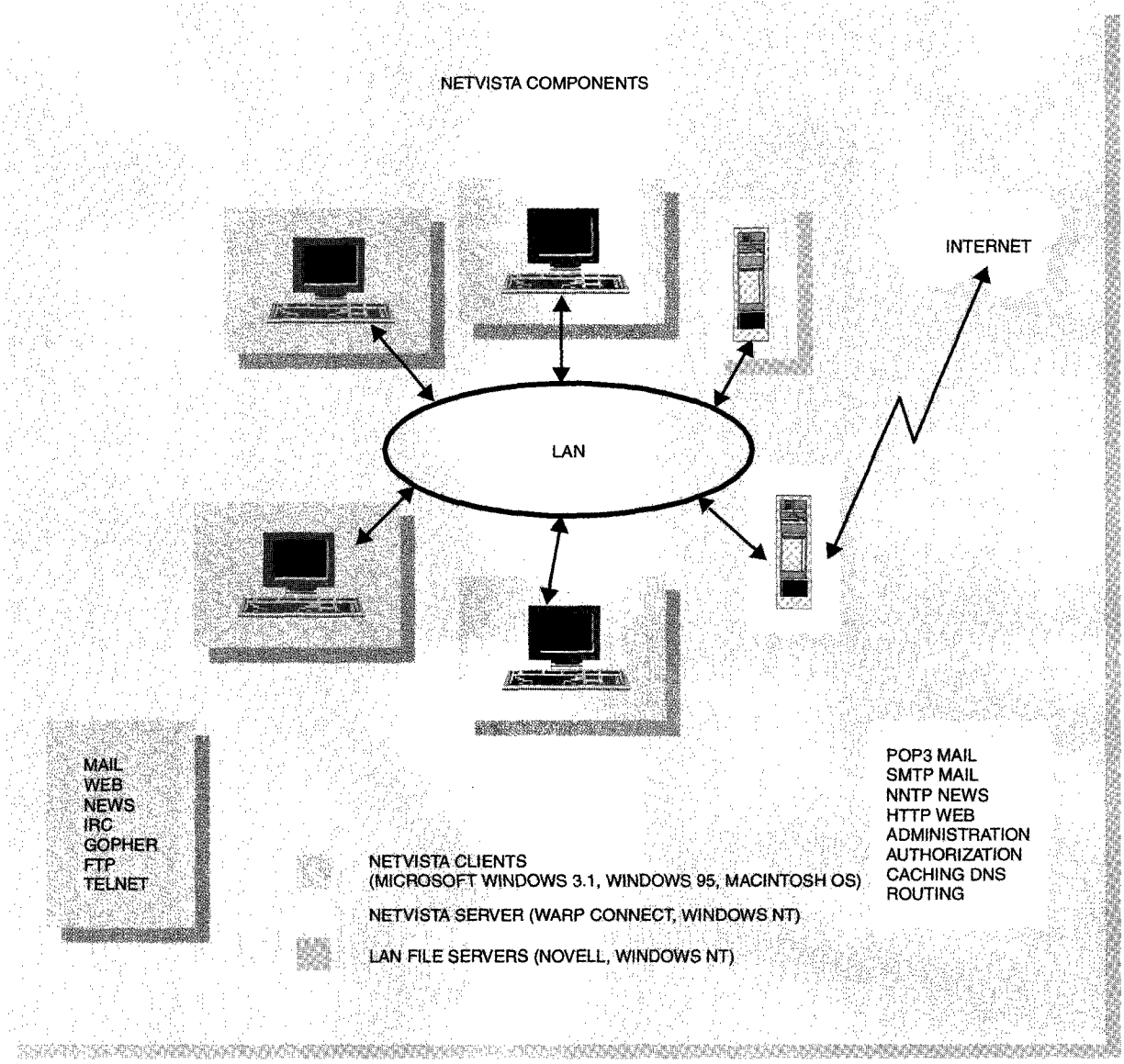
NetVista architecture overview. At the risk of getting somewhat ahead of our story, we offer a brief description of NetVista's overall architecture. We hope that an understanding of the software's main components will assist the reader in following the rest of the discussion.

First, it should be noted that NetVista is a client/server solution. A typical installation will include both a server (running either OS/2 or Windows NT) and an integrated suite of client applications running on Microsoft Windows 3.1, Windows 95, or Macintosh OS workstations. NetVista is typically installed into a school that already has a LAN for purposes of controlling access to and delivering courseware to the workstations. The LAN itself can be simple or quite complex but it must be able to carry both IP (Internet Protocol) packets and (in the case of Novell) IPX** (Internet Packet Exchange) packets. Figure 3 shows a typical installation with a simple LAN.

The LAN file servers supported by NetVista are Novell and Windows NT. Although no modifications are made to these file servers, the NetVista clients and server make requests of them at critical points during user log on and during the processing of mail to tie NetVista into the existing user name space at the school. The NetVista server also (optionally) serves as the gateway to the Internet for all workstations on the LAN. In a low-end installation, the NetVista server is equipped with both a LAN adapter and a modem for this purpose. In higher-end installations, a separate router takes over as the LAN gateway. Note that while all of these components are present in a typical installation, the server and clients can be used separately if the situation warrants. In such a case, the clients behave as standard Internet clients and the servers as standard Internet servers.

The NetVista server is actually a set of server applications, one per supported Internet protocol. E-mail is provided by the usual combination of an SMTP (Simple Mail Transfer Protocol) server and a POP (Post Office Protocol) server. The SMTP server receives mail coming in from the Internet and mail being sent by users from client machines. The POP server manages mailboxes and mailbox contents and

Figure 3 NetVista's architecture consists of a suite of Internet clients (e-mail, news, chat, Gopher, FTP, Telnet, and an integrated Netscape Navigator client) designed for teachers and students, along with a set of servers (SMTP, POP3, NNTP, HTTP) capable of handling Internet traffic and of authenticating users in partnership with the school's Novell or Windows NT LAN server or servers. In addition, an administration client (not pictured) is provided that allows a teacher or computer coordinator to manage the school's Internet access.



provides user access to mail. News, both local and Internet-wide "Usenet" news, is provided by an NNTP (Network News Transfer Protocol) server. The NNTP server receives news articles from the Internet and news posts being sent by users. A Web server, also used as a caching proxy for NetVista clients, an authorization server (for controlling user access), and an administration server (for managing the other NetVista servers) round out the set of servers.

The NetVista client is an integrated suite of applications for viewing and sending e-mail and news, con-

versing on IRC (Internet Relay Chat), browsing the Web, retrieving files via FTP (File Transfer Protocol), and connecting to remote computers via Telnet. The client applications share a number of common interface mechanisms and have been designed to work well together in support of common Internet tasks. They also make nonstandard requests of the NetVista server to achieve improved performance and enhanced functionality. Many of these mechanisms and custom protocols are reviewed later.

Technology for usability

Usability is often viewed as the result of many small design decisions manifested directly to the end user through the application interface. The placement of buttons, the labeling of menu items, and the visual cues directing user attention are elements contributing to usability at this level. At another level, usability is viewed as the result of a good fit between the users' model of a task and the model of the task embedded in the design of the application software. Both perspectives are valid but incomplete. For us, usability includes the ease of system installation, the degree of integration with the larger network infrastructure, the relatively seamless integration with "foreign" applications incorporated into NetVista, the lack of routine maintenance (or alternatively, the automaticity of routine maintenance), and the performance of the client/server package as an ensemble. Iterating in both the field and the lab, we worked in all of these areas to improve the end-user experience. Our work depended on two mainstays for creating a useful, appropriate, and usable solution: a relentless and comprehensive end-user focus, and an object-oriented language and environment (Smalltalk) in which to carry out the development work. In the following sections, we elaborate on these strengths and describe key aspects of our development process and the technology that resulted.

End-user focus. There is no single way to achieve usability in the large, and there is no process that can guarantee good or appropriate design. Recognizing this, the philosophy of the team was that software "seeds" are best sown in the fertile soils of the user environment, with a plan to engage in iteration and adaptation throughout the process. Of the many possible design and development activities that could have contributed to a sustained focus on end users and their environment, our predilection was for informal, "messy," rich, intense methods for understanding our users and their environments; our methods were always rooted in the fact that representative users in representative usage environments were using our software from the beginning and throughout our development process. We drew no diagrams and wrote no scenarios, but we "lived" scenarios by using NetVista for real tasks ourselves, and we got to know our users by spending time installing, modifying, and building code on site and by talking with them about their experiences using the software. We wrote no memos or requirements documents, but we worked together in the lab where design and implementation discussions were continual and pervasive. In this section we outline some of the main factors that contributed to a sustained focus on users in the NetVista project.

An interdisciplinary team. From the start, the NetVista project was unusual in that three of the five team members were cognitive psychologists. One psychologist had expertise and experience with teachers, schools, and education and was responsible for interfacing with our advisory board and with the teachers and technology coordinators in the schools that would become our beta test sites. She was also responsible for creating the user manual, aimed primarily at teachers, for getting started using the Internet in the classroom (this was before other resources, for example, Serim and Koch,[24] were available). Another was a Smalltalk expert and advocate, who not only managed the project, but was responsible for virtually all of the Smalltalk code that created the client and server applications that comprised NetVista. The third was a human-computer interaction expert who took on the role of user advocate during the design process, maintaining the perspective of an end user within design discussions, testing the code continuously by using it for real (and simulated) purposes, and who created the on-line Help system.

This user-oriented subteam was responsible for virtually *all* aspects of the software with which end users would come in contact. At the beginning of the project, their shared, if unarticulated, vision of an easy-to-use, integrated Internet suite served as scaffolding to get the first versions of the client software built. Later, as more pieces were put in place, the mental filters of "Is this useful?" and "Is this appropriate?" became familiar refrains for everything the team did.

The multiple perspectives and expertise of the team, and each person's specialization within a number of key roles, were critical for instantiating our design objectives. Three of the team members virtually lived in the lab during months of intense development and

test activity. This allowed the interplay of user needs and design objectives, on the one hand, and coding infrastructure and capabilities, on the other, to be interwoven in a continuous and fine-grained manner. Design discussions encompassed all of these considerations fluidly, as each of the three team members naturally exercised a particular advocacy (of the user, of the Smalltalk code, and of the underlying operating systems and the communication between it and Smalltalk, respectively). It was in these day-to-day arguments and struggles that a sustained focus on end users was incrementally realized.

Because of our object-oriented implementation, fixing a problem in one place usually fixed it everywhere.

The impact of user-centered design on the implementation. User-centered, object-oriented design implied that the features that users saw were realized by classes and objects in the code; the code ended up being structured based on what the user needed in addition to what the low-level "plumbing" required. As a result, we were able to iterate quickly when we detected a problem with the interface, based on a test subject, or even just to answer a "what if" question. We went through more than 50 significant user-interface designs in our first few months. Similarly we were able to "drop in" alternative implementations with relative ease—such as going from an OS/2 WebExplorer*-based custom-built browser to a Netscape-based browser, or permitting concurrent development of a NetWare**-based user identification management system and a stand-alone NetVista user identification system.

We had the good fortune to work with an extremely adaptable test team from the IBM Endicott Laboratory. They were willing to "throw out the book" on testing and focus on what we really wanted to deliver—bug-free code by a certain date. That meant tracking problems, classifying the ones that mattered, and making sure they got fixed. All other "process" (e.g., a formal process and the stipulated database for tracking problems) was dropped. For example, the test team agreed to use the NetVista news server and client viewer as the vehicle for documenting and responding to problems identified during testing. Each problem was created as a thread in the news reader, and responses, dialog about the problem, and its status (e.g., severity and whether it was open or closed) were entered as contributions to the thread. This allowed an extensive test of the news function under realistic conditions of use. Similarly, the team used the FTP subsystem for code distribution and the mail application for communication. These practices, and other changes that made the testing process more incremental, significantly changed the process to which the test team was accustomed, affecting regression testing as well.

Because our object-oriented implementation was structured around the features that the user actually saw and used, when users or the test team found a problem with an application, the developers could quickly localize the problem, fix it, and put a new version of the code out for the test team to use. It was not uncommon for the testers to have a new version every day—there were actually more versions, but the test team did not want to restart their testing many times each day. More importantly, when a usability problem affected one module (say, e-mail), there was a good chance that it could also affect others (such as news). Our design principles led to an implementation in which common function was represented by a common superclass shared by all objects needing to provide the function. Hence fixing a problem in one place usually fixed it everywhere (or at least pointed to a small set of related subclasses that needed to be fixed).

Our Smalltalk development environment and class libraries also facilitated fixing extremely annoying bugs below the user-interface level—as far down as the TCP/IP (Transmission Control Protocol/Internet Protocol) infrastructure. We were stressing the low-level code in ways that its original developers had never anticipated. If we had not had access to the Smalltalk source for the class libraries, we could never have delivered our work on time. That same environment also permitted a degree of portability and code sharing between different platforms that was uncommon before the advent of Java**. NetVista was developed for Microsoft Windows 3.1, OS/2, Windows 95, Windows NT, and the Macintosh with roughly 90 percent common application-level code among the different platform implementations.

Formative evaluation: NetVista in the field and in the lab. In addition to the system and user testing previously described, the NetVista code evolved from the start in the context of a series of beta test sites that encompassed different users and contexts of use (e.g., elementary vs middle schools vs high schools), and different networking infrastructures. The team was responsible for establishing the relationships with beta test sites and installing and supporting the code once an agreement was reached. An important consequence of having the developers install the initial versions was that they were exposed early to the real-world needs of representative customers and the variety of their networking environments.

In the first beta site, for example, we learned that the usage patterns for Internet clients were not as we had expected. Rather than small groups of users working independently within the classroom, as we had imagined, we found computer labs—large rooms with dozens of computers. Students mainly used the computers as part of computer courses, and during class periods all students did about the same thing at the same time (e.g., reading one or more newsgroups). This meant not only that the load on the NetVista server was very bursty, but that the bursts were directed against a single server (e.g., the news server) at a time—a worst-case scenario for a server. Although unanticipated, this usage pattern appeared in many future beta sites and, once recognized, was obvious. Hence, NetVista's design needed to accommodate sudden and severe increases in activity directed at a single server. A system test tool, described later, was developed to help evaluate each server's ability to cope with bursty usage patterns.

Another lesson we learned from beta-site installation experiences was that many Internet service providers (ISPs) used by schools at that time were inexperienced in connecting entire LANs. In order to get NetVista running at several sites, ISPs had to be contacted and educated to provide services one would assume were already available. For example, several ISPs did not sufficiently understand IP routing: even though a school's modem connection to the ISP was up and running, the school could not reach any external hosts, and no external hosts could reach the school. As a result, the prerequisites for installation of NetVista were carefully written to help guarantee that everything required of a school's ISP was in place and operational before anyone tried to install the product. (There was a selfish side to this as well, since once NetVista installation began, all problems were assumed to be caused by NetVista.)

From our first visit to the model technology school, the team was exposed to its end-user population, sometimes virtually living with the end-user population during installation of beta-test field sites. Each member of the initial five-person team had significant exposure to real end users during the project. The week-long demonstration of the earliest version of the code on a 14-machine LAN at the IBM Schools Executive Conference was a free-form live usability test—with 14 "subjects" at a time, who set their own goals (to explore the Internet, get their e-mail, read a newsgroup) and naturally "thought out loud" and asked questions of the team. We learned a lot, and we taught people a lot about the Internet. We left with a good idea not only of how our software fared, but also of our users' level of knowledge about the Internet and what kinds of help they most needed to get started.

As the project progressed, members of the team spent significant time in the field, either installing at beta sites or troubleshooting when remote access to the Internet server did not resolve the problem. On one of the early trips, the team realized that a workstation-based, LAN-aware set of servers was becoming inevitable. Schools were resistant, we had discovered, to UNIX machines (the solution we had envisioned), and the OS/2 servers that we had been using in the lab had been designed (in certain cases) to support a single user, making adaptation to a LAN difficult. Because the Smalltalk development environment was easily carried from the lab (all our development machines were laptops), the design and coding of the first NetVista servers began in the field. By the end of the visit, the first NetVista servers had been coded and installed.

Often we learned startling things from our field visits, some profoundly inspirational. One of the team members related the following experience:

> When I was installing NetVista into the Hillcrest Elementary School, one of the teachers commented, "You don't have to install it in my room because I will never use it." This teacher was close to retirement and did not see the need for this new technology in the school. Another teacher, who was excited about the Internet, worked with the first for about six months. Some time later, we returned to the school for a follow-up visit. We talked with several of the teachers about their experiences with the software and the Internet. Surprisingly, the older teacher asked if she could talk with us. She was very excited about the software

and what she had learned. She commented, "It is so easy to use. I just click on a button and I get my information," and "It's great to be able to send e-mail to other teachers and look at information on the Internet." Her final comment, which I especially enjoyed, was "I want this for a retirement present!"

Although the beta sites and formal system testing revealed many bugs in the NetVista server, they did not show how well each of the services provided by the server operated under stress. Questions like: "What happens when 100 students try to use NetVista at the same time?" were left unanswered. To answer them, the StressBot Test System was developed. (See sidebar, "Testing the Limits with StressBots"). Essentially, each StressBot simulates a single user repeatedly connecting to a given server: exercising the server, verifying the server's behavior, and then closing the connection. Multiple StressBots were created to simulate multiple users engaging a variety of servers in random patterns. For example, the StressBot Test System was able to simulate the amount of e-mail students would send and receive for a month in about an hour, or the amount of e-mail for a year over a weekend. Without having to wait for problems to develop in the field, the lab-based StressBot system was able to quickly and reliably elicit problems under controlled conditions.

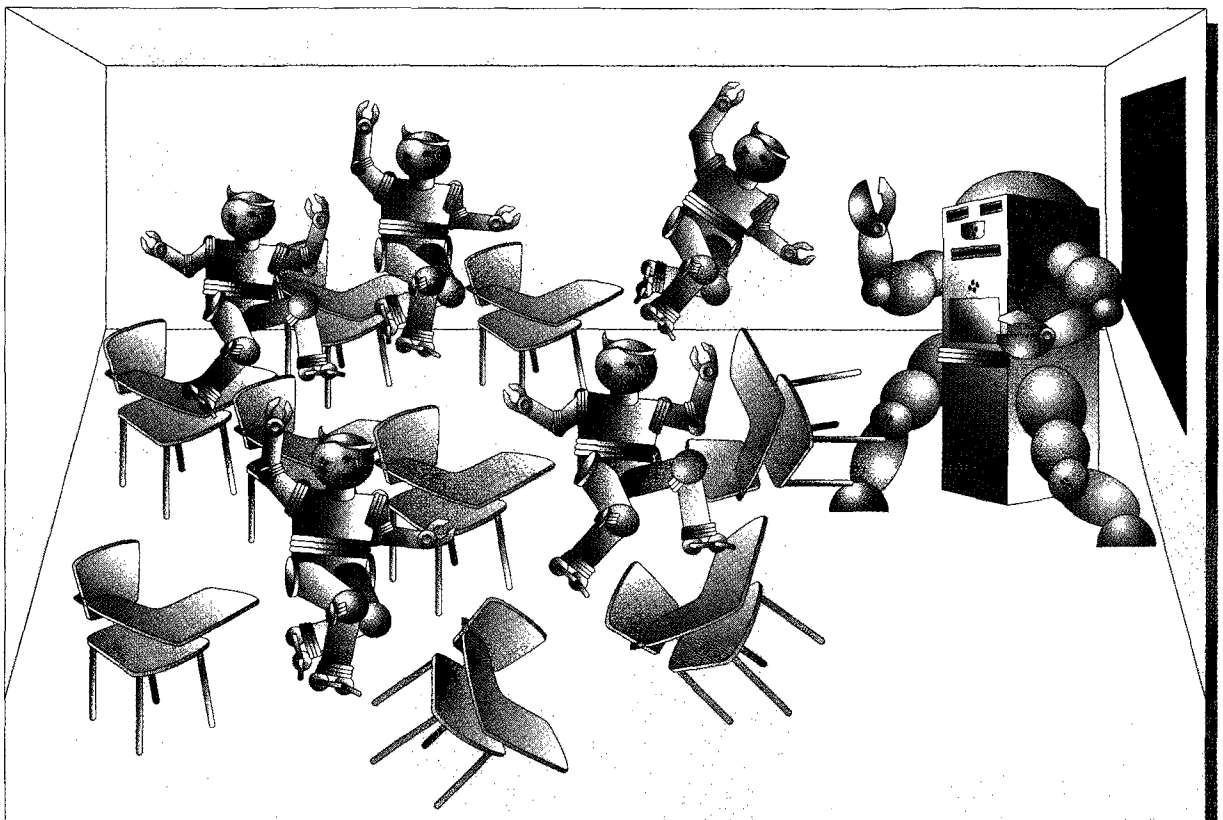
Over a period of several months, we ran thousands of simulations, with two machines generating stress tests for multiple servers. In addition to revealing bugs in the NetVista server code (and even its design), the StressBots also revealed bugs in operating systems and language run-time support. These findings enabled the team to strengthen NetVista's design and implementation before it failed in the field, and, in some cases, to prevent failures that one would not normally expect to encounter in the field (but that we did encounter, given the "creativity" of our field sites, which taught us among other things to be skeptical about our expectations and assumptions).

A flexible coding environment. The coding environment of Smalltalk had many attributes and effects on the project, not the least of which was its ability to support iterations of the design of the applications. What functionality was included, how it worked, and how it was presented to the user were all changed as our understanding of the domain and of the users evolved. For example, we first designed and coded a rather traditional FTP client that displayed both the local and remote directories and contained a button for "transferring" ("getting") a remote file. In transit from a beta site installation, insight and illumination struck: there was no reason that FTP could not look and behave exactly like the more familiar (to our users) Gopher client. A single list of the current directory contents where subdirectories could be opened like Gopher folders would support navigation. A single button for "getting" a file would open up the operating system's standard file dialog so the user could specify where to keep the file. This is, in fact, how Netscape Navigator eventually implemented FTP, but Netscape did not yet exist.

Throwing away an entirely coded and (relatively) bug-free client program is not something most developers want to do on a tight time schedule, and we were no exception. But we did it. Because of the speed with which our changes could be made in Smalltalk, the FTP client was substantially rebuilt by the time the airplane, on which the insight occurred, had landed. The next day in the lab, the new design was scrutinized and its function exercised, particularly in relation to the Gopher client that was its model. We recognized that it was the right decision; it further simplified and unified not only the code, but the user's experience of the Internet. And that was our primary purpose.

Another aspect of the coding environment, important in creating an excellent user experience, is that we always had a running version of the code. Changes in Smalltalk are truly incremental, and it was often only minutes before a design change was provisionally implemented in the already-running application and tried in the context of the current design concern. Some changes were more extensive, or had ramifications in other parts of the code base (and thus were appropriately resisted by the Smalltalk advocate—usually at least until he discovered it was not as difficult to implement as anticipated)—but most often we were able to evaluate proposed changes or improvements rapidly and in the context of the entire suite of applications. This helped us to keep the user's experience in mind throughout the myriad judgments we made during the development process. The same was true, of course, of development work that was carried out at the beta sites. If a user voiced a complaint or made a suggestion, it was often possible to make a change to be tried "on the spot"—helping us to see whether a design change would truly address the issue being raised.

NetVista installation. NetVista installation is potentially daunting. Assuming a properly tuned LAN,



TESTING THE LIMITS WITH "STRESSBOTS"

Software usage in the field is different from its usage in a development or laboratory environment. Users are always doing the unexpected, hardware and software configurations vary widely, and load always exposes hidden design flaws or bugs. We decided to find these problems before shipping NetVista, rather than subjecting our users to them.

"StressBots," short for "stress robots," were used by the NetVista team to run automated stress tests against the various NetVista servers. A stress test involved exercising the server function by simulating heavy usage conditions. To test a server, the tester specified the target host, the services to be tested (e.g., HTTP, SMTP, and POP3), and the number of concurrent users to be simulated. Once specified, the testing system created and launched the appropriate number and kinds of StressBots, and then looped, accepting and displaying returned test results. The NetVista team created several different types of StressBots, targeting various services (e.g., POP3StressBots tested the NetVista POP3 server) or particular aspects of a service (e.g., NewsPostStressBots tested the NetVista NNTP server's ability to handle news article responses).

The role of the stress test system evolved over time. Initially, the team believed its value would be as a simulation of user behavior. In practice, it was most important for efficiently loading and overloading particular servers with certain kinds of input in particular ways, that is, in mounting directed attacks on some aspect of a server's function. This turned the StressBots into a versatile development tool. Failures reported from the field often could be replicated in the lab for debugging purposes. Once a fix had been coded, it could be stress-tested before being deployed in the field. Stress testing also revealed problems that had not (yet) surfaced in the field. These were generally fixed in software updates. The stress-test system was also useful for evaluating performance, such as a comparative analysis of the servers implemented in different languages.

there are still many issues to consider and resolve during this process: client addressing, server identification (domain name server, upstream mail exchanger, news feeder), etc. The usual approach to installing Internet servers involves having the user edit one or more files for each server. Information such as domain name servers and IP addresses must be specified several times, once for each server. Hidden dependencies between files make it difficult for users to complete an installation successfully, assuming they have enough knowledge to understand what they are trying to achieve in the first place. In contrast, our approach was to create special installation programs that would gather the required domain names and IP addresses (once) from the user, and then automatically create the files that were needed to set up the servers. Many iterations of the client and server installers have now made this process reasonably fast and reliable.

For the development of the installation program we adopted an approach similar to that used for client/server development. It was developed in a fast, iterative manner that gave users an opportunity to use the installation program and then provide feedback to the development team. Since TCP/IP and networking is complex, we provided a worksheet to help the user to plan ahead and have available all of the information needed before beginning the installation. We tried to have the installation program provide intelligent default values for as many information fields as possible. For example, when the server installation program was invoked, the installer function would read all the local networking files to determine as much as possible for the installation. It would find out the machine name, domain name, IP address, primary domain-name server address, and secondary domain-name server address. If the server was already connected to the Internet, it would query the domain name servers (DNSs) for the DNS names. Because this information was obtained automatically, the user did not have to type the unfamiliar and complex host names and numerical addresses. Thus potential typing errors were prevented.

By user request, the installation program looped through the display of installation information panels. The user could set up, and review, the information as many times as desired before committing to the installation. All information was saved in a ".dat" file so that the user could exit the installation at any time and restart without losing information. This allowed users to seek their own level of comfort before beginning the installation "for real." It also allowed an installation to be interrupted, and missing information obtained, without restarting the entire process. The administration client program installation and the NetVista client installation program were developed in a similar manner. All the information was gathered "up front" to configure the application, then the installation program would create and modify files as needed. Once again, these are small design details, but they add significant value when users deviate from the "ideal" installation procedure, which is to say much of the time.

In Canada, to make the installation process even simpler, a complete server was built in the lab, containing all the defaults and server/client software, and copied to a master disk. When the installation team was ready to install NetVista at the customer site, they would copy the master disk to the hard disk of the NetVista server machine. Then the installation program would be invoked to customize the system for the school. With this approach, servers for complete Internet access can be built and customized in approximately one-half hour. The client installation software is copied over the network to the local Novell server and installed on the server. A complete installation for both clients and servers takes approximately 1 to 2 hours.

Server administration. No matter how easy NetVista is to use, it is still something new for a school to deal with. From the administrator's point of view, it is yet another server with another set of functions (some of them quite complex) to be learned and managed. Our experience with the beta sites showed that often the administrator was a teacher who had volunteered to manage the school's technology infrastructure. These inescapable facts motivated much of our server-side work. By maximizing the automaticity of maintenance and the degree of integration with the larger LAN context, we attempted to make the server invisible. And by creating a simple, integrated view of the server suite, we attempted to make each administrative task as straightforward and error-free as possible.

We decided early that since NetVista was going to be installed in schools with LANs, we would design it to take advantage of the preexisting Novell or Windows NT user databases. One approach would have been to provide a database import function that allowed the administrator to clone an existing database during setup. But since school populations are fairly transient (at least a portion of the student body leaves at the conclusion of each school year) it made

more sense for the NetVista server to query the database when decisions had to be made. Consider the case of incoming mail. When presented with a mail item for a user in its domain, the server needs to determine if the user is valid and able to receive the mail. In NetVista, this is done by querying the Novell or Windows NT server (or servers) at the point of decision. If the user is known by the Novell or Windows NT server the mail is accepted. And, if a mailbox does not already exist for the user, one is created. Or take the case of mail being picked up by a remote user (i.e., one not currently on the LAN and hence one not already authorized by Novell or Windows NT). The password presented by the POP client is checked against the Novell or Windows NT server to determine whether the mailbox is opened or not.

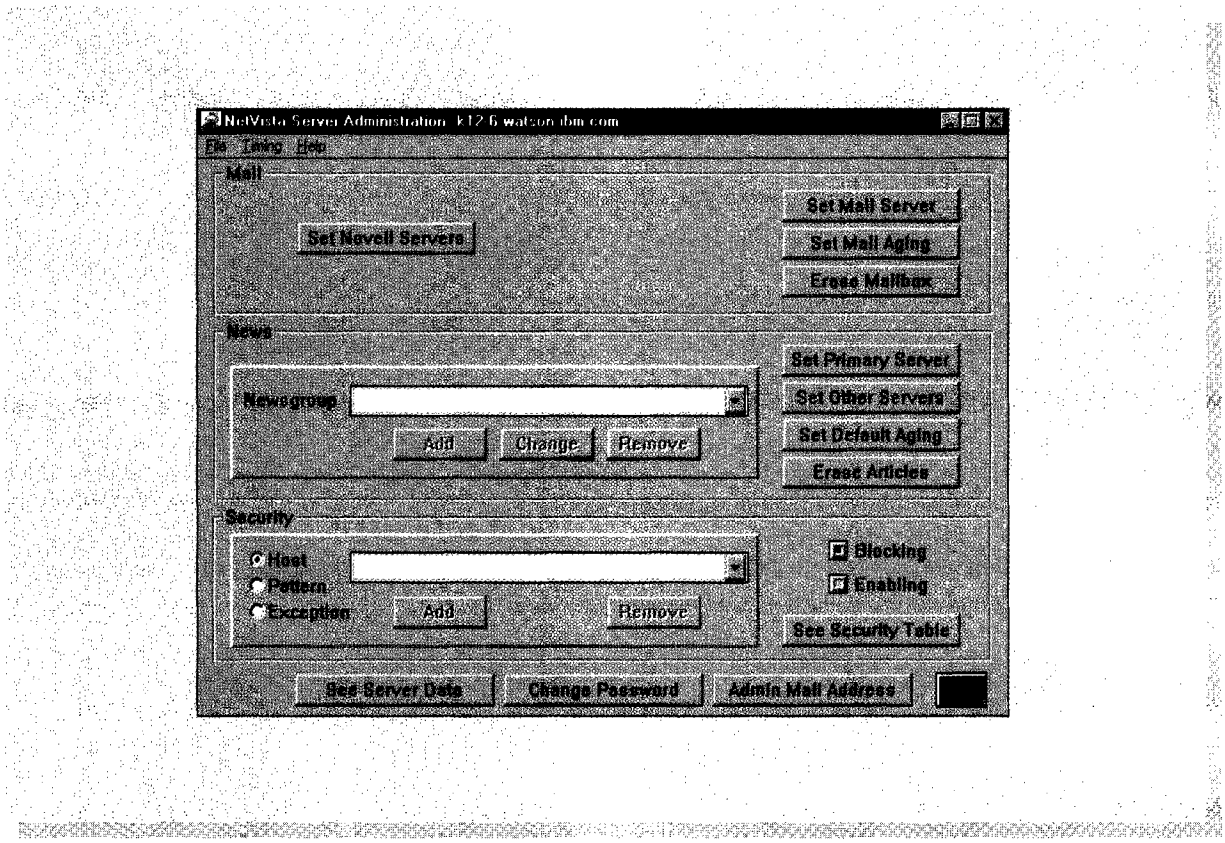
Once a mailbox is created, system resources are consumed. One early design did not reclaim these resources when the mailbox was emptied. At the suggestion of our early users, we modified NetVista to automatically remove empty mailboxes. Thus, at the end of the school year, there is no associated maintenance. Because the user has been removed from the Novell or Windows NT server, no more mail will be received. And since any existing mail will be automatically purged after the server-specified holding period, the server will eventually remove all traces of the user from storage. This is another seemingly small thing. But to an already overburdened administrator, it is seen as highly desirable.

Occasionally, the administrator needs to change some aspect of the NetVista server. For example, the administrator may want to set up a new discussion group, or change the blocking filters, or view the storage consumed by user mail. To support these tasks we created an administration server and an administration client (Figure 4) that provided a single, coherent view of all the server functionality. Consider the potential complexity of adding a new discussion group using typical management schemes (involving fairly complex syntax, a text editor, and multiple files). NetVista provides a single screen that allows all aspects of the new newsgroup to be specified, including whether responses can be posted to the group, which other servers will receive these posts, and whether the group is to be protected by a password. In addition, NetVista verifies that server names can be resolved before they are added. Hence, a common problem (e.g., news articles building up in the server because it cannot get rid of them to an invalid server) will be detected before it affects the server. More subtly, the existence of a single view of the suite of servers forced us to consider each new server feature from the perspective of the administrator. Would a proposed addition be readily incorporated in the administration client? Would it be manipulated like other server features, using common mechanisms, or would it require a new set of administrator skills? These questions helped to keep us focused on simplicity for administrators throughout the development of the server suite.

Other factors supporting an end-user focus. Finally, there were interesting synergies that formed around the factors previously discussed. As the project progressed, these provided an intangible but real sense of what would work in the design and what would not, what was important enough to struggle for and what was not. NetVista's design seemed to coalesce in a way that made new ideas and proposals easier to evaluate as time went on. In addition, there were some unique characteristics of the project that wielded important influences, such as the vision of early champions of the project from the IBM K–12 environment, the energy boosts provided by our first experience demonstrating the NetVista code, and the enthusiasm and dedication of a special marketing representative in Canada[25] who was instrumental in supporting many of the early beta and services-offering sites and served as a high-bandwidth information conduit between the users and our team.

All of these factors yielded excellent guidance for keeping the best, most user-oriented functions and losing the rest, separating the user-worthy wheat from the chaff, so to speak. The integration of the clients helped us present Internet pointers (such as URLs and host sites) in a uniform way, and made it possible to provide special integrative functions. For example, we provided a function so that users could point at a URL, host-site name, or e-mail address in any text window (e.g., mail or news). Clicking within such a string would produce a pop-up menu offering functions specific to the string. Clicking on an e-mail address would offer functions such as "Add this address to my nicknames list" or "Send a mail message to this address." Clicking on a URL would invoke a pop-up menu allowing the user to open it in the Web browser, add it as a bookmark, or forward it to someone else in a mail message. (See Figure 5.) When we thought about adding functions to a particular client, we always did so within the context of all of the clients to which users would be exposed.

Figure 4 The NetVista administration client. From this screen, all NetVista server functions can be monitored, controlled, and modified. Note that the management of the user mailbox names is redirected to the Novell server or servers, set through the "Set Novell Servers" button.

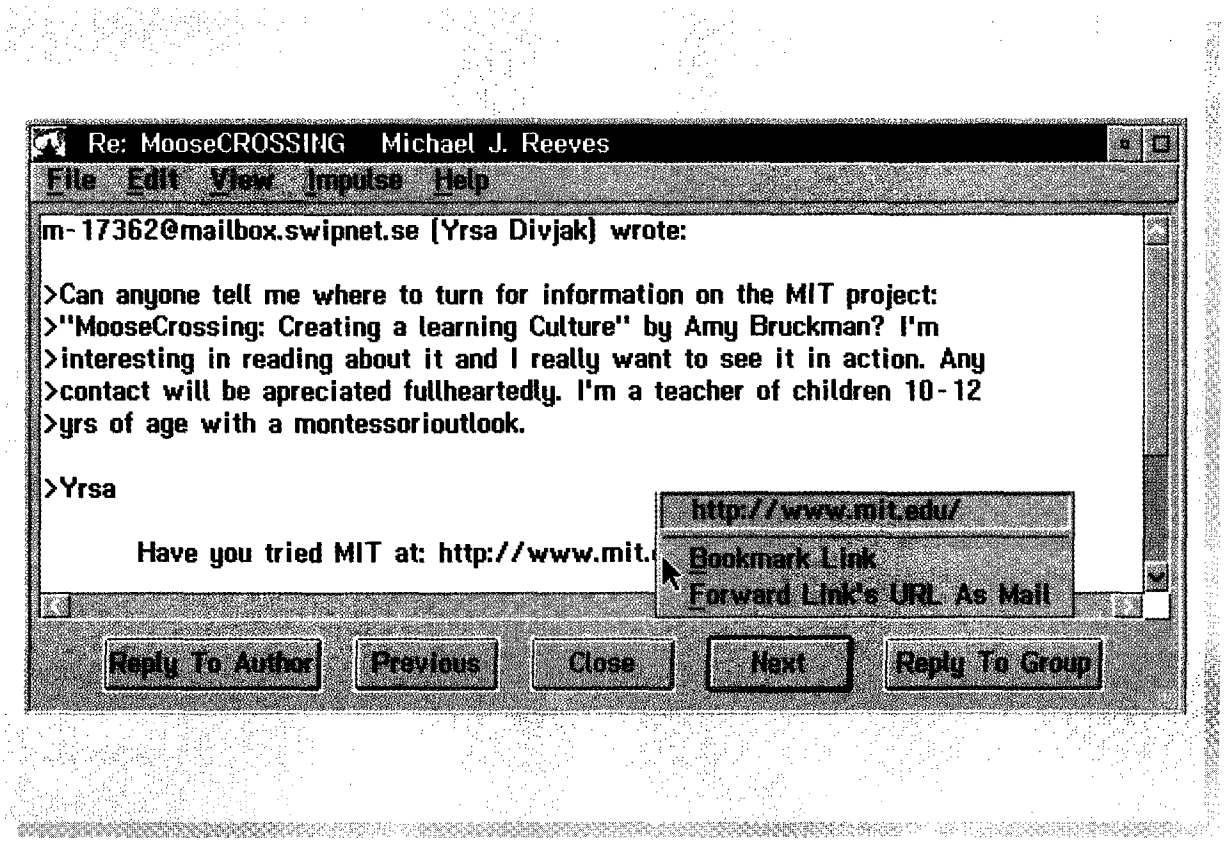


The abstraction and refinement of common code in the underlying object-oriented framework was also an impetus for common appearance and behavior at the user interface. From a design point of view, we wanted similar functions to appear and act the same way. From a coding point of view, this was often achieved by running the same code, but having the different objects for which the code was invoked provide appropriate data. Thus, a two-way interplay between design objectives and the structuring of the code took place throughout the project. This process, which depended on the recognition of common function, flexibility in the coding environment, and the ability to reuse code throughout the various clients, was a key aspect in managing the tension between rapid development and a high-quality user experience.

To summarize, the main factors contributing to a focus on end users included the expertise and integration of perspectives of an interdisciplinary team, an ongoing, continuous process of formative evaluation in the field and in the lab, and a flexible coding environment. In the next section, we switch our focus to the characteristics of this coding environment and the development of NetVista's technical infrastructure.

Object technology base. A sustained focus on users allowed the NetVista team to identify what worked well and what needed to change. But knowing what needs to change is not enough to ensure that a product will be usable and desirable. We know of projects in which the problems inherent in a design are discovered in time but the cost of fixing them in a non-

Figure 5 Pointing at a URL in text and the resulting pop-up menu. The pop-up menu allows the user to open the Web browser to the URL, to bookmark it, or to forward it to someone else in a mail message. This was done *without* preparsing the message or translating it to HTML.



malleable development environment is prohibitive. We know of projects in which the resources to fix problems are available but every change (applied as "patches" to the existing code) increases the size and decreases the robustness of the implementation. In our own case, *much* needed to be changed as NetVista evolved, even though we started with a good set of requirements and an initial design that was more often "on target" than not. Things that worked well in the early versions were found to be out of place in the growing application suite. Whole applications were developed, only to be discarded and replaced. New functions were introduced that needed to be applied coherently across multiple protocols and data types. In order to accommodate this level of change we needed a flexible development environment, with powerful tools for structuring and restructuring our code. We needed an environment in which changes actually *increased* the stability and integrity of the code base. In this section we summarize the important aspects of the coding environment in which we worked, and look at NetVista from the inside out.

NetVista's clients and servers were developed almost entirely in Smalltalk, an object-oriented language and development environment. Smalltalk provided two key advantages. First, by supporting rapid incremental development, it allowed us to explore our design *in situ*, testing and modifying running applications on the Internet. Many more options can be meaningfully explored when tests can be conducted in minutes to hours rather than days to weeks. Code is also of higher quality when it can be tested within a running system as soon as it is written (rather than waiting for the elements to be brought together late

in the cycle for a "system" test). Second, by providing tools for structuring our code as objects within a hierarchy of classes, Smalltalk allowed us to evolve client and server *frameworks* that captured more and more cross-application behavior as well-tested, inheritable code. While it is impossible to prove that NetVista could not have been built without these capabilities, it is clear to us that without them, our small team would not have attempted the task.

The NetVista client/server application frameworks. Frameworks are essentially abstract applications. They capture the common behavior of a range of related applications and make this behavior immediately available to any application developed as a specialization of the framework. As such, they are powerful both for structuring code and in supporting the reuse of that code.[26] Frameworks have been developed in many domains including two of interest to us here: support for network protocols[27] and support for graphical user interfaces.[28] The NetVista code frameworks contributed to quality in numerous ways (consistent behavior across components, enhanced integration across applications, code compactness, and ability to redesign around performance bottlenecks, to name a few). Most importantly, the frameworks allowed us to steadily improve the common code that was used by all clients and servers. Thus, over time performance and reliability improved *as* new function was added. Iterating over the design in search of simplicity, integration, and usability similarly improved the quality of the code. And reflecting on the properties of the emerging framework suggested further simplifications and unifications in the design as seen by the user. Some examples of the interplay between usability and framework evolution follow.

Naming and connecting to Internet servers—Naming things (servers, URLs), and making use of those names later, lies at the core of effective Internet use. Within the NetVista framework, a common mechanism for resolving names and making socket connections is inherited by all clients. This allows all clients to support a rich model of naming without the overhead of creating it. From the users' perspective they need only remember a name of a bookmark or a nickname to reconnect to any URL or any server (including those, like chat, in which there is no URL involved). The common inherited "resolver" code maps this name through the user's personal nickname list and hierarchical bookmarks and expands it appropriately. Common connection logic is also inherited by all clients. This allows a consistent and simple model of connection progress to be reflected to the users across all applications. This logic also made it easy to add the capability to block certain hosts, domains, and URLs, since it was placed in the abstract client superclass and then inherited for use by all clients.

Finding things—Once communication is established with a server, the user is often confronted with the task of finding a particular thing in a list of things. It became apparent that a common mechanism for finding things would be advantageous. Within the client framework, this was implemented by having all "findable" objects implement a specialization of the inherited asSearchableString method, thereby returning to the sender of this message whatever the object wanted to offer as its target string. This allowed a common search mechanism to operate across newsgroups, news posts, mail lists, FTP directories, Gopher elements, bookmarks, IRC channels, IRC nicknames, etc. New objects could participate in this scheme by implementing a single method while retaining the flexibility to determine a meaningful target for the search.

Coping with errors—Once something is found on the Internet, it is commonly downloaded to be viewed or manipulated. Downloaded files can be large and, at least in the K–12 environment, disk space is often limited. Common file exception handling allows all clients to fail *nicely* when an attempt is made to write to a disk where space is not available. This is unified across clients and across platforms by a single mechanism that is used pervasively.

Achieving adequate performance on small machines—An important aspect of usability is performance; without good performance, even the best design cannot support a good user experience. A well-structured code framework tends to be small. Identifying and factoring out common behavior leads not just to better code but to *less* code. Since poor performance on small machines is often the result of memory contention and swapping, code compactness leads directly to better performance.

Leveraging client/server integration. The evolution of NetVista client and server applications brought both better performance and better function. This was particularly true when we (judiciously) extended the mail and news protocols for sites running both the client and server suites. We provide two examples of these extensions.

Opening a mailbox with many items on a POP server is slow; an inefficient request/response cycle has to be repeated for each mail item on the server. To accommodate very large mail drops (where the mail is retained on the server) we created a new server mechanism that returns the mailbox overview (i.e., sender, date, and subject for all messages) in an already-parsed form. When a school installs both the NetVista mail client and the NetVista mail server, this greatly speeds the opening of mail. Of course, when the components are installed separately, the NetVista mail client works correctly with other POP servers, and the NetVista POP server works correctly with other mail clients.

NetVista allows private newsgroups to be created and modified easily (e.g., a password can be added or removed, posting can be turned on or off as needed). But NNTP provides no mechanism for keeping a client's view of newsgroup properties synchronized with the server's view (other than by generating error messages). This can lead to situations where, for example, posting has been turned off for a newsgroup, but users are still able to post to the newsgroup because the client is "out of sync." Another private protocol was created to address this synchronization problem.

Novell integration. After talking with several school administrators, it became clear that we needed to use their existing Novell server name space. Schools already had procedures for entering user names and passwords into the system for their Novell accounts. When logging into Novell, the student or teacher had already entered a user name and password. For both usability and security reasons, we did not want to have NetVista ask again for the name and password. Also, using the same user name on the Internet would allow users to have a single identity on both the Novell server and the Internet.

NetVista is designed so that when it starts, it queries Novell for the "logged-on" user information. From this it can determine the user name of the person logged on, which becomes the Internet user name. This process also avoids the problem of a person logging into Novell as one person, and then entering a different name for the Internet, i.e., attempting to "spoof" (pretend to be) another user for e-mail. The Novell user name also becomes the Internet mailbox name when e-mail is received. As described previously, the NetVista server queries the Novell server (or servers) to determine whether incoming mail is addressed to a valid user. If the user name is not found, the mail is rejected. Thus a one-to-one mapping of Internet and Novell user names is maintained. This arrangement is much appreciated by the schools, since it prevents site administrators from having to define and maintain a separate name space for Internet access.

Making it Macintosh. One of our final challenges was to create a true Macintosh version of NetVista. For many reasons, the first target for NetVista was Microsoft Windows 3.1, a GUI (graphical user interface) platform small enough to fit into school-sized Intel-based computers. The fact remained, however, that many schools were on a Macintosh base, and we wanted to support these customers as well.

Although significant system and user-interface differences exist between Microsoft Windows and Macintosh OS, both are based on the "desktop" metaphor, utilizing windows, icons, menus, and pointers to accomplish the same goals. Given the proper system support, a mapping between the two platforms was tractable.

Without a common computer language between Microsoft Windows and Macintosh OS, supporting NetVista on two different computer systems (hardware and software) would have been prohibitively expensive. Rewriting in a different language would have been very costly in time and person resources. Operating system and language differences would have required accommodations in the internal design. Further, it would have created a maintenance nightmare for future NetVista development and support. Each change would need to be planned for, implemented, tested, and tracked in two separate implementations. A common programming language between Microsoft Windows and Macintosh OS was required.

Fortunately, a common programming language solution did exist at the time, although the solution was far from perfect. As discussed above, NetVista is implemented in the Smalltalk programming language. By its design, Smalltalk is a high-level language based on a "virtual machine" implementation. The details of the underlying hardware and operating system are (mostly) hidden from the Smalltalk programmer. Our Smalltalk vendor, Digitalk, Inc., was marketing what we were looking for: Smalltalk/V** Macintosh. To the extent that Smalltalk/V Macintosh was identical to Smalltalk/V Windows, the cross-platform issues would have already been solved for us by our colleagues at Digitalk.

Inevitably, there were significant differences between the Smalltalk implementations on Microsoft Windows and on Macintosh. The Smalltalk syntax is quite compact. Many components in the Smalltalk environment are well-specified, system independent, and therefore could be coded identically for each platform. Outside this common ground, however, the situation was more complex. Developed from different original code bases, in different parts of the United States, and designed to accommodate the style and philosophy of the underlying platform, differences were understandable, but not good news. The major software components upon which NetVista relies are the window system, TCP/IP networking, the file system, and interprogram communication. Of these, only the window system and the file system were addressed in the cross-platform Smalltalk definition. In order to bring NetVista to the Macintosh, the other differences had to be reconciled as well.

In addition, with the introduction of Apple Guide in System 7, the model of on-line help in the Macintosh became significantly different from the Microsoft Windows and OS/2 platforms. Apple Guide eschewed graphically rich "mini-tutorials" and descriptive information for help systems in favor of seriously task-oriented, step-by-step procedures presented in small windows viewable from an application. Accordingly, the on-line help system for NetVista was completely reconceptualized and reimplemented to support this model.

Conclusion

NetVista was shipped in June, 1996, with a second release in November, 1996. At the time of the June, 1996, release, NetVista had been in beta test in eleven schools for approximately two years. From the beginning of the project, the five-person core team had created the software on Microsoft Windows 3.1, OS/2, Windows 95, Windows NT, and Macintosh OS. As mentioned previously, earlier versions of the code had been released as a Limited Availability Services Offering in 1994 (for Microsoft Windows 3.1), and as beta software on a compact disk and over the Internet in 1995 (for OS/2).

As members of the CHI (computer-human interaction) community, and long-standing advocates of the value of usability engineering and iterative design, it was somewhat surprising to reflect on our own design process and see how little it incorporated some of the activities typically prescribed, particularly more structured and formal analyses of users, tasks, and context.[29] It is important to point out, however, that this information was not missing from our design process—rather it was provided via a continual and rich process of immersing the team in the user's environment and developing the software in the context of real usage. It is hard to imagine that it could have been otherwise, or that more formal representations of these users and their contexts could have served us better. In the final analysis, although we believe we could have learned useful things from more formal analyses, such as usability tests in the lab, the process we followed was essential for grasping the crucial requirements of a complex project and melding them into a useful and appropriate product.

Similarly, feedback from students and teachers in the field using NetVista has been informal and continuous throughout the project. We have not yet visited some of the new schools using NetVista, but we are delighted to see some of the work they are doing on the Internet. (See, for example, http://cses.scbe.on.ca/index.htm and http://www.lbe.edu.on.ca/bonavent/welcome.htm, which have both won site awards.) We are also gratified to see the emergence of district-wide licenses for NetVista, including the Province of British Columbia, Canada, which has licensed its 1700 schools to use NetVista. Success with our users and in the marketplace is our ultimate yardstick. We are interested in carrying out measurements of effectiveness and satisfaction with the NetVista software, but in the meantime, changes and refinements to the software continue to be driven from ongoing informal reviews and conversations with our users.

Our experience and reflection leads us to expand our notion of what a small team can achieve, and to emphasize particular aspects of our project as conducive to creating software rapidly that is focused on usability in the large. We state the lessons we have learned in general terms in the hope that they can be more easily applied by others, and we hope that the foregoing discussion has given the reader some idea of how these abstractions were realized in practice within the NetVista project:

- Begin a design process by immersing principal team members in the users' environment. Continue this exposure throughout the life of the project.
- Create a collaborative, reflective design process that has significant representation of different perspectives. Respond to the insights produced by these perspectives.

- Learn to live with design questions. Give your code and your understanding of the design space time to evolve together. In a supportive coding environment, this need not translate into longer development time. Be prepared to handle radical revisions if necessary.
- Maintain an intense focus on simplification. No matter how sophisticated your users, seek a design that does not exceed the inherent complexity of the problem the design is addressing.
- "Live" your users' scenarios to the greatest extent possible. Use your developing system for your own work wherever applicable.
- Seek continual informal feedback from users or representative others, even if you can afford a more formal usability testing program. Better yet, have representative users use your code from its earliest incarnations.
- Be willing to ignore "common knowledge" (i.e., conventions or what others say you must do), and focus on what really needs to be done.

Although our project was largely unable to take advantage of conventional usability measures and activities, we believe in the value of a test suite of user scenarios and tasks to drive design in the direction of greater usefulness, appropriateness, and usability. At the same time, we wonder whether practitioners of "user-centered design" sometimes become complacent—satisfied if a project incorporates standard usability practices, whether or not they actually succeed in achieving usability in the large. A paper representation of user tasks, or even an elaboration of particular usage scenarios will almost never be as effective as putting developers in the user's environment to meet and interact with the users around the software. We are committed to reaching for a design ethic deeper than common GUI elements, and a process more meaningful and more discovery-oriented than can be easily prescribed. In this quest, an enduring end-user focus and a coding environment that makes it possible to respond to the kinds of changes that focus can engender are paramount.

Each development project is unique, and we do not mean to suggest here any procedure or recipe for success. We do know that there seemed to be unmistakable signs that we were on the right track: for example, the fact that our work was truly collaborative and engaged multiple points of view, and that the effect of iterations and introducing new elements was visible in the simplifying contractions of the evolving code framework. Other aspects were important as well: we worked with detailed knowledge of who our users were (with the corresponding inconvenient realities that knowledge forced us to address), and we had the luxury of designing a significant portion of the school's Internet solution, one that could respond to the entirety of the environment in which the software would exist.

Our experience has led us to realize that the iteration in this project had a different focus than the one we normally think of in traditional usability practice. We iterated not in the sense of climbing incrementally toward usability objectives, but in terms of discovery: What is this software about? What function belongs? How can we make this simpler? As designers, we sought to reveal to ourselves, and then through our design to our users, the most basic and important dimensions of using the Internet. As developers, we sought to reform and gain insight in our understanding of the domain of Internet protocols, clients, and servers, and the relationship between Smalltalk and the underlying operating systems with which we worked. What emerged from this process is not only a particular product from a particular team, but collaboratively owned expertise and an object-oriented framework for implementing networked applications that can be extended in new directions in the future.

The potential for extending our work to new domains or problems is important to us from a research perspective. It has taught us the value of developing a whole solution within a niche, and then turning attention to how it might apply or be leveraged in new domains. We have often seen work that takes the approach of developing technology meant for "everyone." However, we suspect that as networked applications grow more sophisticated and capable, many of the most interesting efforts will find their origins in particular domains, with particular users, where a real-world problem urges upon us a new understanding of computing in a networked world.

Acknowledgments

We thank Tom Erickson, John Thomas, John Karat, and three anonymous reviewers for helpful comments. We also thank the members of the larger NetVista family: Rink Bingham, Jonathan Brezin, Jerry Chwazik, Gary Cole, Norm Cox, Don Daria, Dave DeSantis, Scott Engleman, Stu Feldman, Armando Garcia, Pat Goldberg, Ambuj Goyal, Jan Jackman, Jeff Jaffe, Merwyn Jones, Carl Kessler, Brian Mackay, Petar Makara, Bruce McClellan, Dave McQueeney, Bruce Nelson, Lori Neumann,

Bob Petti, Bill Rubin, Dave Smith, Vicki Spirito, Bill Stanton, Pat Sueltz, Mike Sutherland, John Vlissides, Leslie Wilkes, David Wood, and Carol Young.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Microsoft Corporation, Apple Computer, Inc., Netscape Communications Corporation, X/Open Co., Ltd., America Online, Inc., Novell, Inc., Sun Microsystems, Inc., or Digitalk, Inc.

Cited references and notes

1. C. Danis and J. Karat, "Technology-Driven Design of Speech Recognition Systems," *Designing Interactive Systems (DIS'95)*, G. Olson, Editor, ACM Press, New York (1995).
2. J. D. Gould, S. J. Boies, S. Levy, J. T. Richards, and J. Schoonard, "The 1984 Olympic Message System: A Test of Behavioral Principles of System Design," *Communications of the ACM* **30**, No. 9, 758–769 (September 1987).
3. T. K. Landauer, *The Trouble with Computers: Usability, Usefulness, and Productivity*, MIT Press, Cambridge, MA (1996).
4. J. Lai and J. Vergo, "MedSpeak: Report Creation with Continuous Speech Recognition," *Human Factors in Computing Systems: The Proceedings of CHI'97*, S. Pemberton, Editor, ACM Press, New York (1997), pp. 431–438.
5. A "Web-year" is only three months long.
6. F. P. Brooks, "No Silver Bullet—Essence and Accidents of Software Engineering," *IEEE Computer* **20** No. 4, 10–19 (April 1987).
7. Telnet is an application used for logging on to a remote computer.
8. Mosaic was the first publicly available commercial Web browser.
9. *Computers & Education* **24**, No. 3 (April 1995). Special issue on education and the Internet, W. A. Kellogg and D. W. Viehland, Editors.
10. R. Eurich-Fulcer and J. W. Schofield, "Wide-Area Networking in K–12 Education: Issues Shaping Implementation and Use," *Computers & Education* **24**, No. 3, 211–220 (April 1995).
11. *Gopher* is an application that presents information from Internet servers as menus of folders and documents; *news* refers to bulletin-board-style discussions over the Internet on a particular topic; a *chat* is a multiway text-based conversation with others on the Internet; *FTP* (File Transfer Protocol) is a way to find and transfer files from (or to) remote computers.
12. E. Soloway and R. Wallace, "Does the Internet Support Student Inquiry? Don't Ask," *Communications of the ACM* **40**, No. 5, 11–16 (May 1997).
13. W. A. Kellogg and J. T. Richards, "The Human Factors of Information on the Internet," *Advances in Human-Computer Interaction, Volume 5*, J. Nielsen, Editor, Ablex, Norwood, NJ (1995), pp. 1–36.
14. P. Heckel, *The Elements of Friendly Software Design: The New Edition*, Sybex, Inc., Alameda, CA (1992).
15. B. Laurel, *The Art of Human-Computer Interface Design*, Addison-Wesley Publishing Co., Reading, MA (1990).
16. B. Laurel, *Computers as Theatre*, Addison-Wesley Publishing Co., Reading, MA (1991).
17. D. Norman, *The Psychology of Everyday Things*, Basic Books, New York (1988).
18. B. Shneiderman, *Designing the User Interface: Strategies for Effective Human-Computer Interaction (Second Edition)*, Addison-Wesley Publishing Co., Reading, MA (1992).
19. B. Tognazzini, *Tog on Interface*, Addison-Wesley Publishing Co., Reading, MA (1992).
20. See http://www.ibm.com/ibm/hci/guidelines/design/principles.html.
21. *User-Centered System Design*, D. Norman, Editor, Lawrence Erlbaum Press, New York (1987).
22. J. M. Carroll and W. A. Kellogg, "Artifact as Theory-Nexus: Hermeneutics Meets Theory-Based Design," *Human Factors in Computing Systems: The Proceedings of CHI'89*, K. Bice and C. Lewis, Editors, ACM, New York (1989), pp. 7–14.
23. J. M. Carroll, W. A. Kellogg, and M. B. Rosson, "The Task-Artifact Cycle," *Designing Interaction: Psychology at the Human-Computer Interface*, J. M. Carroll, Editor, MIT Press, Cambridge, MA (1991), pp. 74–102.
24. F. Serim and M. Koch, *NetLearning: Why Teachers Use the Internet*, O'Reilly and Associates, Sebastapol, CA (1996).
25. Fred Jennings, IBM Canada.
26. R. E. Johnson and B. Foote, "Designing Reusable Classes," *Journal of Object-Oriented Programming* **1**, No. 2, 22–35 (June/July 1988).
27. H. Hüni, R. Johnson, and R. Engel, "A Framework for Network Protocol Software," *OOPSLA '95 Proceedings*, ACM Press, New York (1995), pp. 358–369.
28. M. A. Linton, J. M. Vlissides, and P. R. Calder, "Composing User Interfaces with Interviews," *Computer* **22**, No. 2, 8–22 (February 1989).
29. D. Hix and H. R. Hartson, *Developing User Interfaces: Ensuring Usability Through Product and Process*, John Wiley & Sons, Inc., New York (1993).

Accepted for publication September 12, 1997.

Wendy A. Kellogg *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598 (electronic mail: kellogg@watson.ibm.com).* Dr. Kellogg, a research staff member, holds a Ph.D. degree in cognitive psychology from the University of Oregon. Her research interests include human-computer interaction, the Internet, and virtual communities.

John T. Richards *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598 (electronic mail: jtr@watson.ibm.com).* Dr. Richards manages the Niche Networking department at the Thomas J. Watson Research Center. His research interests include object-oriented application development and human-computer interaction. Dr. Richards holds a Ph.D. degree in cognitive psychology from the University of Oregon.

Calvin Swart *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598 (electronic mail: cals@watson.ibm.com).* Mr. Swart is a senior software engineer whose work has included object-oriented toolkits, video subsystems, and Internet technology.

Peter Malkin *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598 (electronic mail: malkin@watson.ibm.com).* Mr. Malkin's work has included mobile robotics, multimedia object-oriented databases, and network design. He is an advisory software engineer.

Mark Laff *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598 (electronic mail: mrl@watson.ibm.com).* Mr. Laff, a research staff member, has been affiliated with the Thomas J. Watson Research Center since 1960. His research has ranged from fractals to operating systems and from video editing to VLSI (very large scale integrated) circuit logic design. He continues to try to make computers do things that are useful for people.

Vicki Hanson *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598 (electronic mail: vlh@watson.ibm.com).* Dr. Hanson manages the K–12 Networking and Collaborative Learning department at the Thomas J. Watson Research Center. Her research interests include language and reading, software tools for performance-based assessment of students, and professional development for teachers. Dr. Hanson holds a Ph.D. degree in cognitive psychology from the University of Oregon.

Brent Hailpern *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598 (electronic mail: bth@watson.ibm.com).* Dr. Hailpern manages the Internet Technology organization at the Thomas J. Watson Research Center, which includes workflow, Internet server performance, Internet software for K–12 education, electronic mail, intelligent agents, and antivirus software. Dr. Hailpern holds a Ph.D. degree in computer science from Stanford University.

Reprint Order No. G321-5661.