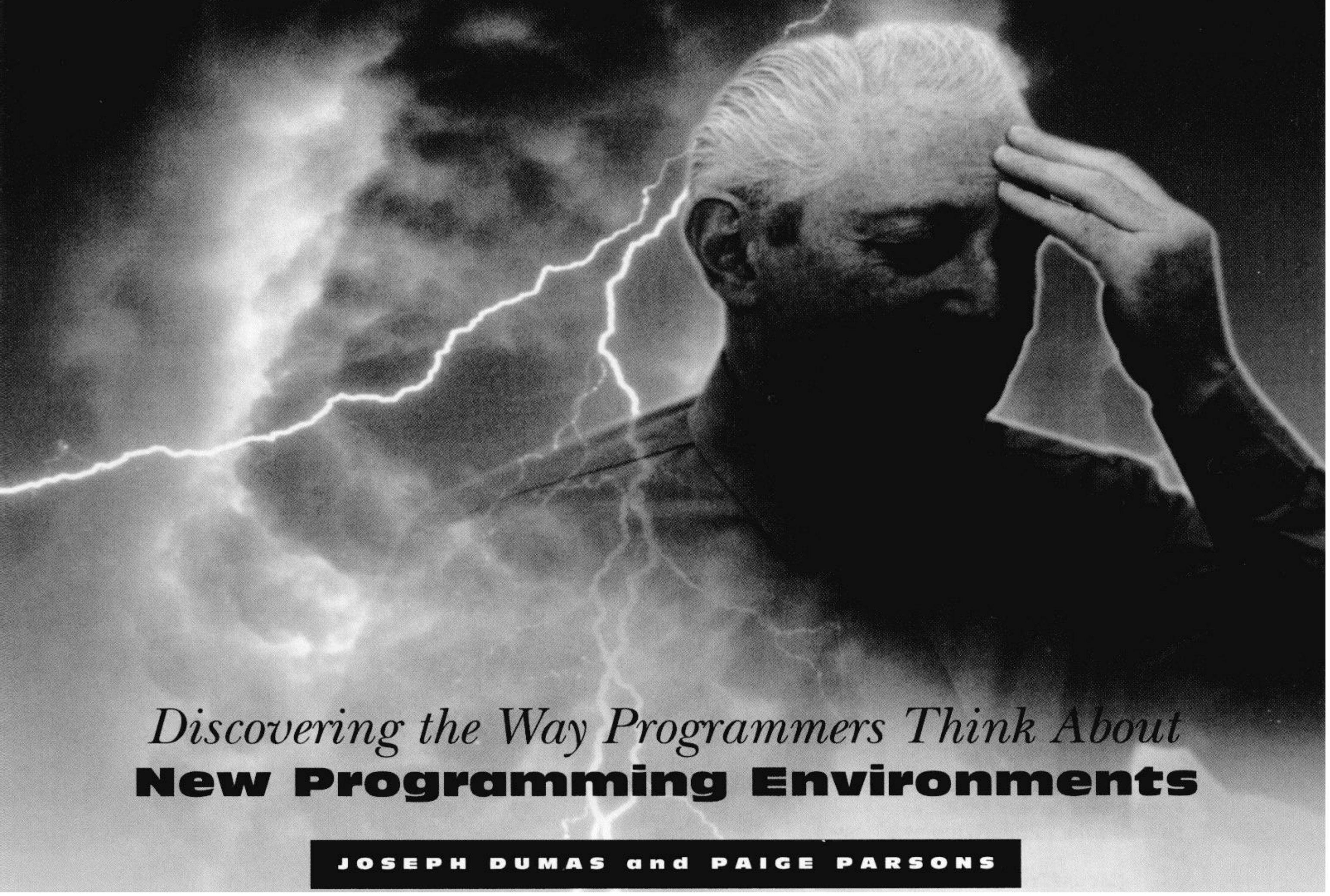
An interesting usability study of a prototype development environment for the Dylan programming language is presented here. This study's purpose is to determine just how close the prototype is to developers. New approaches to source code organization and to the relationship between the environment and the application being developed are introduced. An assessment of how effectively the prototype conveys these innovations to Dylan developers is also given, followed by some proposed improvements.



Discovering the Way Programmers Think About **New Programming Environments**

JOSEPH DUMAS and PAIGE PARSONS

This article describes the first empirical user test of a prototype for a new programming environment. The programming environment was designed to work with Dylan™ [3], a new object-oriented programming language. The prototype's user interface showed the menu structure of the environment, and provided a set of tools for performing a limited number of programming tasks. For example, users did not compile code with the prototype, but they did browse and modify the code of an existing program.

The test had two objectives. The first was to "debug" the usability of several critical parts of the user interface. We wanted to find out how easy or difficult it was for programmers to learn and use the prototype's source code organization, cross-development model, and window-linking mechanism. The second objective was to discover how programmers viewed the capabilities of this new programming environment. Specifically, we wanted to see if the organization of the user interface suggested new ways to think about the tasks involved in programming. In order to meet these objectives, we modified the methods commonly used to conduct a user test by having a test administrator probe the subjects' thinking at key points in the test.

Our usability test uncovered several important usability problems and allowed us to discover how programmers were thinking about the new environment. This has confirmed our belief that usability testing is a valuable method for exposing usability problems and exploring ways that programmers think about programming.

Figure 1. A typical source code file. Related statements are grouped spatially. Whitespace and ASCII characters are used to delineate related definitions and logical breaks.

Background

When programmers create programs, they work with a variety of tools in addition to the programming language itself. These tools include debuggers for diagnosing errors in the running application, browsers for searching and editing source code and objects, and compilers for recompiling the code once the changes have been made. The tools and the programming language used with them are commonly

```
//   UMouseTrackBehavior.cp
//   Copyright © 1992 by Apple Computer, Inc. All rights reserved.
//   Kent Sandvik DTS
//   This file contains the basic TMouseTrackBehavior member functions
//   Version Info:
//   <1>    khs    1.0     First final version
//   <2>    khs    1.0.1   Fixed a memory leak in TMapApplication::GetSleepValue()

#ifndef __MOUSETRACKBEHAVIOR__
#include "UMouseTrackBehavior.h"
#endif

//   Initialize needed parts for the MouseTrackBehavior modules
#pragma segment AInit
pascal void InitMouseTrackBehavior()
{
      if (gDeadStripSuppression)
      {
             macroDontDeadStrip(TMouseTrackBehavior);
             macroDontDeadStrip(TTrackWindow);
      }

      RegisterStdType("TTrackWindow", 'ttrk');
}

//   Close the remove the single swallow application when closing the floating window
#pragma segment AClose
pascal void TTrackWindow::Close()
{
      gMouseTrackWindow = NULL;                    // signal that it's OK to open one again

      gApplication->RemoveBehavior(fBehavior);     // Get rid of the TSwallowBehavior in gApplication...

      inherited::Close();
}

//   Get the pointer to the single swallow behavior which we need when removing it
#pragma segment ARes
pascal void TTrackWindow::GetOriginatorBehavior(TSwallowBehavior* behavior)
{
      fBehavior = behavior;
}
```

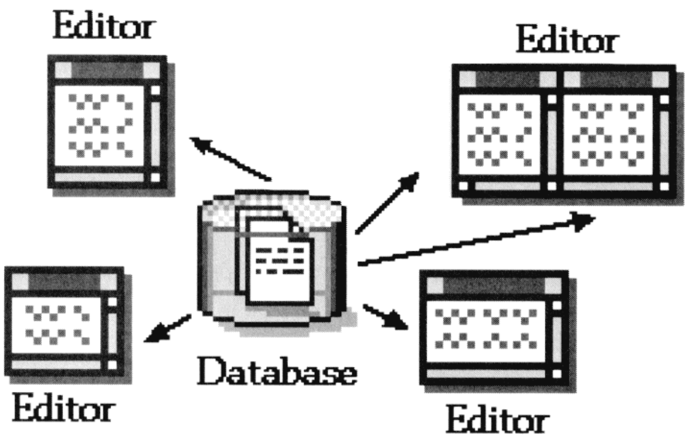


Figure 2. An object-oriented storage model. In an object-based system, definitions are stored in a database, and definitions are edited in separate editors.

referred to as the development "environment."

Software manufacturers now offer environments containing tools and programming languages that are designed to work together and have a common user interface. From the programmers' perspective, the tools provided with the environment are often as important as the language itself. For example, the success of Visual Basic [10] has been largely attributed to its programming tools, not the appeal of programming in the Basic language. Competition among programming environments has evolved from a focus on functionality alone, that is, offering more tools or more sophisticated tools, to competing on usability as well as functionality. While programmers are clearly computer literate, they, like any end users, want tools that provide both new capabilities and ease of use. The evolution in the programming environment marketplace is no different from the trends we have seen with other software tools, such as spreadsheets and word processors.

Complicating the picture, however, is the realization that as tools and users become more sophisticated, major productivity gains are accomplished only through changes in the way that work is thought about and done. A challenge for the programming language developer is to provide a user-friendly environment while helping programmers think differently about the way they create programs.

The Prototype

The prototype used in our test was an early version of the Apple Dylan environment. The environment was designed to develop applications in Dylan, a new dynamic object-oriented language.[1] This goal of the Dylan project is to provide programmers with improved productivity while at the same time enabling the practical delivery of small, fast applications. This is accomplished through a variety of language and development environment features. Our usability test was limited to an exploration of the development environment, so we will describe only the ways in which the development environment differs from traditional commercial environments. We did not test the usability of the syntax or semantics of the Dylan language.

Source Code Organization

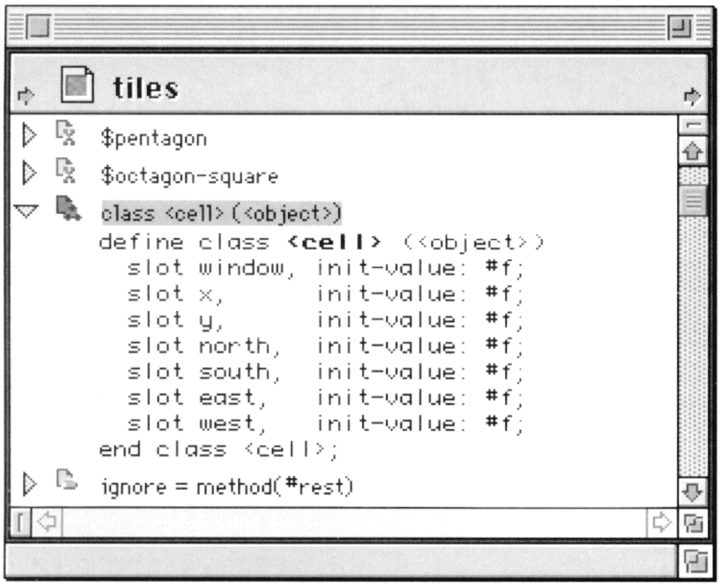
Most development environments in wide use today are file-based. In file-based systems, related code is placed in a file, or in a group of files with related names. Within each file, proximity within the editor is also used as a smaller grouping mechanism. To make it easier to find sections or individual definitions, whitespace and lines of repeated ASCII characters are often employed as delimiters between related definitions (see Figure 1). The basic building blocks of a file-based system are tightly coupled sections of code fragments in a file.

The file-based approach has ramifications for saving and compiling as well. When a portion of the file is modified, usually the whole file is marked to be saved and compiled. In addition, the ordering of definitions that the user sees in the editor is the same as the ordering saved to disk. The storage model is reflected in the user's view of the code.

In contrast, the environment that was prototyped manipulates code at the granularity of definitions rather than files. Individual definitions are the primary building block and unit of storage. An object-oriented database is used to store and retrieve definitions (see Figure 2) and individual editors are used to modify each definition. The definitions are represented as individual objects to the user (see Figure 3). These objects, known as source records, can be directly manipulated in the user interface, or their contents can be edited as text.

Because the basic building block is a definition, individual definitions can be saved, edited, or other-

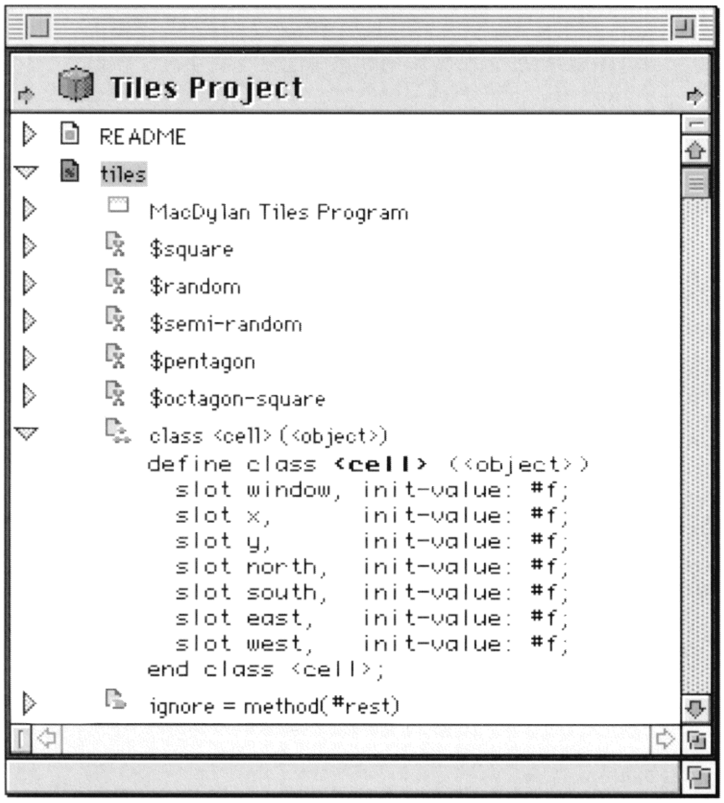
[1]Object-Oriented Dynamic Languages are commonly referred to as OODLs. An OODL is defined as a language that is object oriented, has automatic memory management, supports dynamic linking and incremental development, and provides self-identifying objects.



```
$pentagon
$octagon-square
class <cell> (<object>)
  define class <cell> (<object>)
    slot window, init-value: #f;
    slot x,      init-value: #f;
    slot y,      init-value: #f;
    slot north,  init-value: #f;
    slot south,  init-value: #f;
    slot east,   init-value: #f;
    slot west,   init-value: #f;
  end class <cell>;
ignore = method(#rest)
```

Figure 3. A source record is a representation of a definition. The window shows source records for two variables, one class, and one method. The class source record has been expanded to show its definition. These objects can be directly manipulated in the user interface, or edited as text.

Figure 4. A source container is a folder-like mechanism for grouping and ordering source records. The window shows two source containers. The second, "tiles", has been expanded to show its contents.



```
README
tiles
    MacDylan Tiles Program
    $square
    $random
    $semi-random
    $pentagon
    $octagon-square
    class <cell> (<object>)
      define class <cell> (<object>)
        slot window, init-value: #f;
        slot x,      init-value: #f;
        slot y,      init-value: #f;
        slot north,  init-value: #f;
        slot south,  init-value: #f;
        slot east,   init-value: #f;
        slot west,   init-value: #f;
      end class <cell>;
    ignore = method(#rest)
```

wise manipulated without affecting the other definitions in the system. The location of the definition in the database is not under the control of the programmer, and is independent of the display order used within the development environment. Because the storage model does not imply a user-level ordering, an independent user-controlled grouping mechanism is provided. This grouping mechanism is called a source container (see Figure 4). A source container is a folder-like user-interface object that holds any number of individual source records (definitions) in a user-specified order.

The Development Cycle

Traditionally, programmers working with static languages, such as C [9] and C++ [12], go through an edit–compile–link–debug cycle. They make changes, recompile a large segment of the program, relink it, test, and try again. Unfortunately, this approach has the disadvantage of requiring large recompiles when small changes are made. Dependency systems such as Make [5] help to reduce the scope of the recompile, but in most cases, changes to a single definition still require a fair amount of unrelated code to be recompiled.

On the other hand, programmers working with dynamic languages, such as Lisp [11], Smalltalk [7], and Dylan have a different development cycle. Their edit-compile-debug cycle is much more incremental because only modified definitions need to be recompiled. In addition, these environments have traditionally provided other timesaving features that facilitate rapid prototyping, such as automatic memory management. In Lisp and Smalltalk, this dynamic approach has the disadvantage of blurring the distinction between the development environment and the application under development. Applications built with these environments usually include multi-megabyte images containing the compiler and other portions of the development environment.

The Interactive Cross-Development Model

The Apple Dylan development environment has two goals. First, it should allow programmers to build their programs interactively and incrementally, as in Lisp and Smalltalk. Second, it should generate small, fast programs with a clear separation between the development and runtime, like static development environments. To simultaneously achieve these goals, the development environment employs an interactive cross-development model (see Figure 5).

The development environment and the runtime application execute as separate application processes. The development environment process manages a database that stores source code, compiled code caches, and debugging information. It is responsible for incrementally downloading compiled definitions and libraries

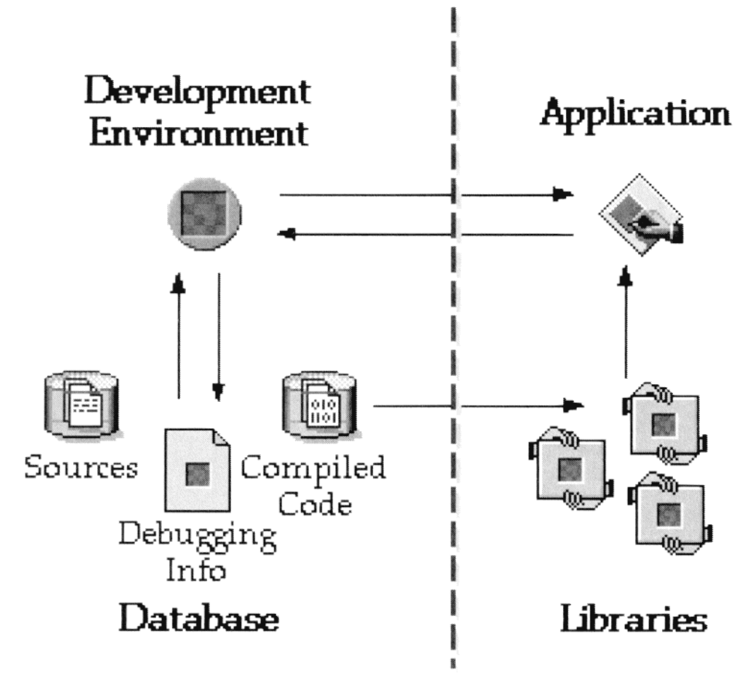


Figure 5. Interactive cross-development model. Source code, compiled definitions and debugging information are stored in a database. The development environment downloads compiled definitions and libraries to the application under development. When development is complete, the links between the developed application and the development environment are severed, resulting in a small deliverable application.

to the separate application under development. The application under development is executed, even during debugging, as a separate process. These two processes are usually running on the same machine, but this is not required, enabling remote debugging and development in distributed systems.

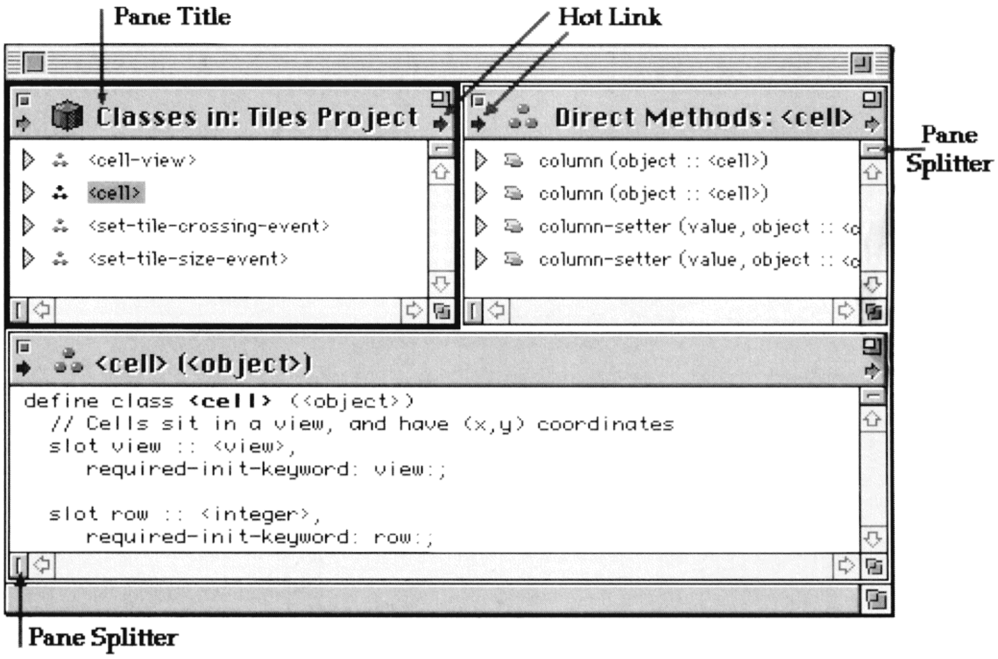
Communication between these concurrent processes is very different from the typical C and C++ model. In most static languages, the code is either being edited in the development environment, or the application is running. Interactively modifying an independent application process requires a different conceptual model of program development.

Multi-Paned Linked Windows

Another attribute that makes developing in Apple Dylan different is the environment's flexible pane-based[2] window system. Pane-based window systems have been successfully employed in a variety of different development environments such as Object Master [1] and Smalltalk-80 [6]. Pane-based browsers are a powerful tool because they allow the programmer to have linked groupings of related code. One pane may show a class hierarchy, while another shows the source for a selected class in the hierarchy. Development environments that make use of paned browsers usually provide several configurations that are employed for a variety of special purposes, such as inspecting groupings of related methods, attributes of a class, or relations between methods and classes. Although these pre-configured browsers are useful, they do not allow much customization or tailoring to an individual programmer's browsing needs.

In the environment we tested, windows may contain any number of panes, each displaying a user-configurable type of information. The user can change the type of information that is shown in the pane by select-

[2]Panes are like nested windows; the only limitation is that panes within a window may not overlap.



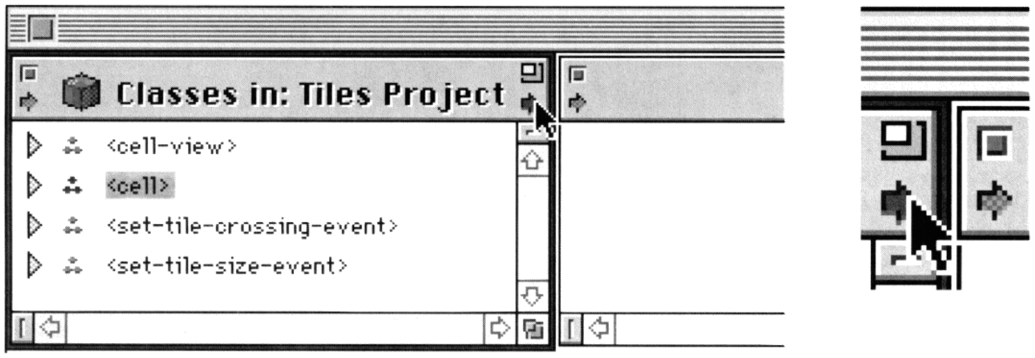
```
<cell-view>
<cell>
<set-tile-crossing-event>
<set-tile-size-event>
```

```
column (object :: <cell>)
column (object :: <cell>)
column-setter (value, object :: <c
column-setter (value, object :: <c
```

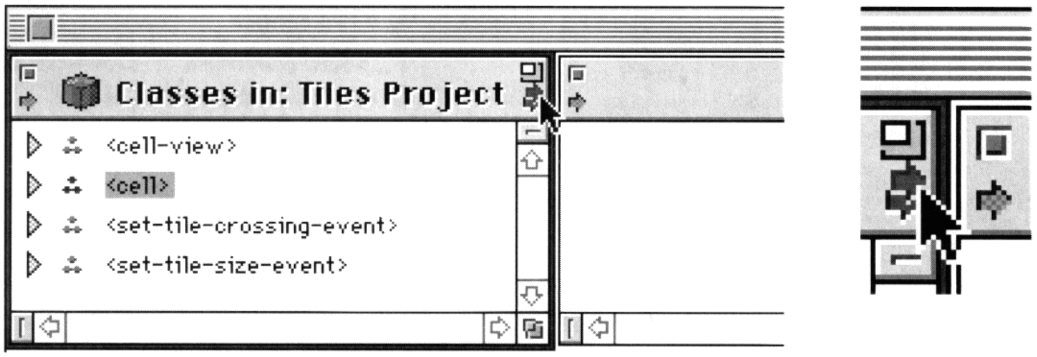
```
define class <cell> (<object>)
  // Cells sit in a view, and have (x,y) coordinates
  slot view :: <view>,
    required-init-keyword: view:;

  slot row :: <integer>,
    required-init-keyword: row:;
```

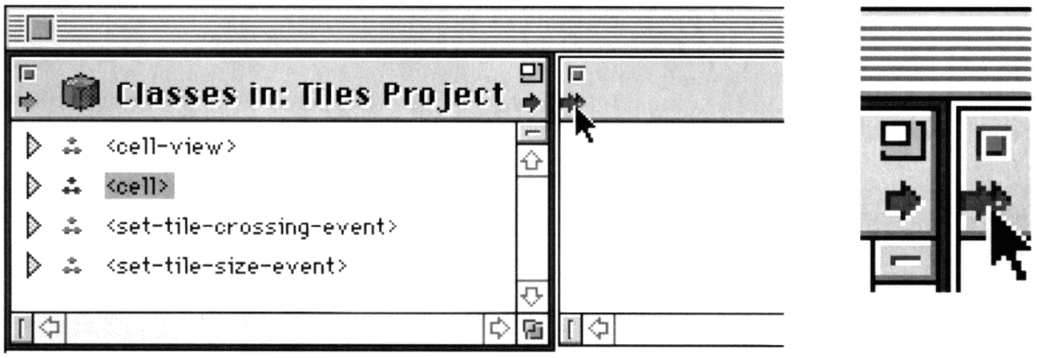
Figure 6. A multi-paned browser. Each pane has controls for splitting, resizing, zooming, scrolling, and closing.


Classes in: Tiles Project
<cell-view>
<cell>
<set-tile-crossing-event>
<set-tile-size-event>

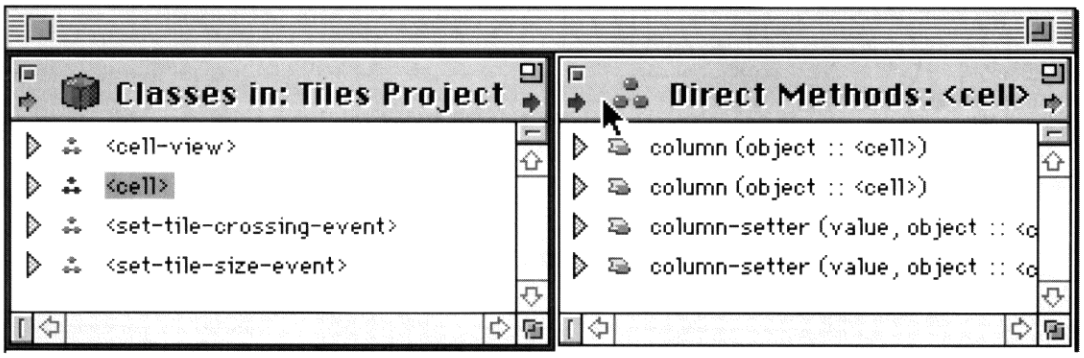
7a


Classes in: Tiles Project
<cell-view>
<cell>
<set-tile-crossing-event>
<set-tile-size-event>

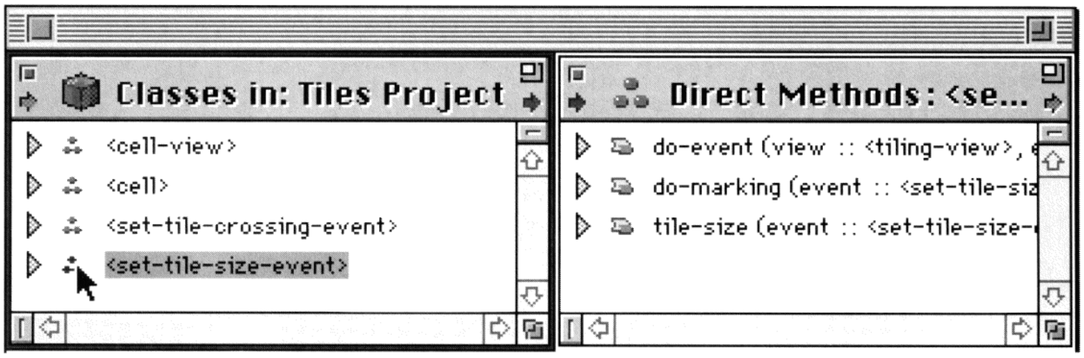
7b


Classes in: Tiles Project
<cell-view>
<cell>
<set-tile-crossing-event>
<set-tile-size-event>

7c


Classes in: Tiles Project
<cell-view>
<cell>
<set-tile-crossing-event>
<set-tile-size-event>
Direct Methods: <cell>
column (object :: <cell>)
column (object :: <cell>)
column-setter (value, object :: <c
column-setter (value, object :: <c

7d


Classes in: Tiles Project
<cell-view>
<cell>
<set-tile-crossing-event>
<set-tile-size-event>
Direct Methods: <se...
do-event (view :: <tiling-view>,
do-marking (event :: <set-tile-siz
tile-size (event :: <set-tile-size-

7e

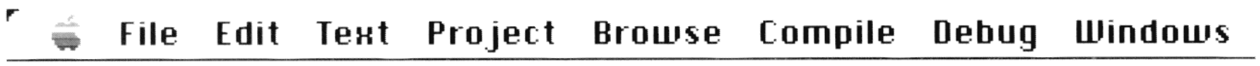


Figure 8. The Apple Dylan menubar. At the beginning of the usability test, users were asked what they thought would be in each menu.

ing from a list of aspects available via a menu. Examples of aspects include "contents," "direct methods," "source code," and "references to." In addition, each pane has its own title bar and set of controls for resizing, zooming, scrolling, splitting, and closing. For example, Figure 6 shows a browser with three panes. The upper-left pane is displaying the "classes" aspect of a project. The upper-right pane is displaying the "direct methods" aspect of the class <cell>. The lower pane is showing the "source code" aspect of the class <cell>.

The user creates new panes with the horizontal and vertical split bars located on the edges of the scroll bars. The size of the panes is controlled by placing the pointer on the border between two panes, and then clicking and dragging.

In addition, the window system contains a mechanism for creating links between panes within a single window or across multiple windows. To associate information across panes, a user creates a *hot link.* A hot link connects two panes so that selecting an object in the first pane causes the object to expand to show related objects or editors in the second pane. For example, in Figure 6, the upper-left pane is hot-linked to the upper-right pane and also to the lower pane. The information shown in the right and bottom panes is updated based upon the selection in the left pane.

Figures 7a–7e show the process for hot-linking panes. To link two panes, the user grabs (clicks and drags) the output (right) arrow of the first pane (Figures 7a, 7b) and drops it (releases the mouse button) on top of the input (left) arrow of a second pane (Figure 7c). Both arrows become red to indicate the link (Figure 7d). The pane on the left now controls what is displayed in the pane at the right. In addition to examining the arrows, the user can deduce the behavior of these panes empirically by selecting items in them and watching what happens in the other panes. Notice that <cell> is highlighted in the left-hand pane and its direct methods are displayed in the right-hand pane (Figure 7d). When the selection changes in the left pane, the contents of the right pane are updated, in this case showing the direct methods of the class <cell-view> (Figure 7e).

Figure 7. Hot-linking panes
• *Figure 7a.* The user clicks and holds onto the output arrow of the left pane.
• *Figure 7b.* The user drags the left pane's output arrow toward the right pane's input arrow.
• *Figure 7c.* The user drops the left pane's output arrow on the right pane's input arrow.
• *Figure 7d.* The hot link is established. The arrows that form the connection become red to indicate the link. The hot link causes information about items selected in the left pane to be shown in the right pane.
• *Figure 7e.* When the selection changes in the left pane, the contents of the right pane are updated.

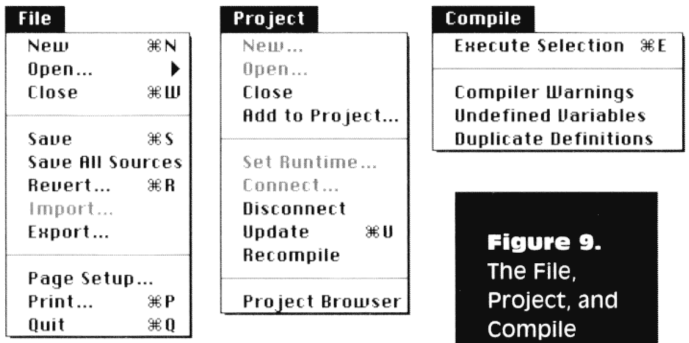


Figure 9. The File, Project, and Compile menus.

The Test Method

In order to understand how users were thinking about the Apple Dylan saving model, cross-development model, and pane-linking mechanism, we performed a usability test. Usability testing is an empirical method for finding the most serious usability problems with the user interface to a product [8]. There are five characteristics of every usability test [4]:

1. The focus is on finding usability problems.
2. The participants in the test come from the population of people who will use the product.
3. The participants perform typical tasks that the product was intended to support.
4. The test administrators observe the test participants and record their performance.
5. The test administrators diagnose the usability problems they record and propose solutions to the problems.

In a typical usability test, test participants "think out loud" while working alone. The test team watches and records key events. This method is assumed to simulate situations in which users normally learn to use a new application or to simulate the condition in which the user has no support from colleagues or a customer support phone line.

Having the test participants perform tasks alone would have allowed us to probe the usability problems with the programming environment prototype and to accurately measure the programmers' performance, such as their task times. But their working alone would provide less insight into how the participants were thinking and learning about the new environment. Studies of the "think out loud" technique have shown that most of the statements participants make are simply verbal descriptions of the actions they are taking. Such descriptions would uncover usability problems, but would not help us to understand how the participants viewed the menu structure or what users thought they could do with the environment.

We chose to use a specialized usability testing method known as "active intervention" [4]. In this method, the test administrator sits with the test participant, watching what is happening. The test participant thinks out loud as he or she works. During tasks that are designed to probe the usability of the interface, such as attempting to link panes, the administrator simply watches the participant without comment. However, at prearranged points, the administrator asks questions of the participant that are intended to reveal how he or she thinks about the environment and to probe for additional information whenever the participant does or says something insightful.

The Test Participants

We recruited eight professional programmers for the test. Recent studies of usability testing have shown that 80% of usability problems are uncovered by five test participants. Ten participants uncovered 90% of the problems, but 100% of the serious usability problems [14, 16]. All eight participants had been programming professionally for at least the past two years, were familiar with Apple's System 7 [2], and had some experience with object-oriented programming. In screening the test participants, we excluded programmers who worked at developing programming environments because they might have some specialized knowledge that most programmers would not have. In addition, we excluded programmers who were heavy users of Lisp or Smalltalk, two dynamic programming environments. We chose to exclude them because we were concerned that their previous exposure to dynamic environments would have a noticeable impact on their comprehension of how this environment worked.[3]

The Procedure

With the exception of the active intervention role of the administrator, we followed a fairly standard usability testing format for all of the testing subjects. As preparation, we first had the participants complete a short background questionnaire and sign a confidentiality agreement. We then had them read a brief (two-page) high-level overview document that described the capabilities of the development environment. We also informed the subjects that there were observers on the other side of a two-way mirror who were interested in how the software performed, and that the session was being videotaped. We explained to the participants that they would be performing a set of predefined tasks using a new programming environment, and made it clear that the focus of the test was on how the product performed, not on how they performed. As the final step of preparation, we gave the subjects a short training session on how to think out loud.

For the actual test, we had the subjects perform a set of tasks and also answer questions. The questions were posed by the test administrator, who was sitting in the test room with the participant. The tasks we had them perform were designed to probe the usability of the user interface. For example, to test the users' understanding of the source code organization, we had programmers textually edit and save changes to code from a program we had constructed. We also had them browse the code to look for a variety of definition types, such as classes, methods, and comments. We asked participants to add and remove individual definitions from a project. In another set of tasks we explored the hot-linking mechanism and multi-paned windows. Early in the test session, we exposed the programmers to a browser with three linked panes. Later in the test, we removed the browser and asked the programmers to recreate it. We gave them a one-paned browser and asked them to "separate this window into three panes and link them together so that the browser behaves like the one we worked with earlier."

The test administrator's questions were designed to gain insight into the participants' understanding of what they could do with the environment. For example, we began the test with three questions probing programmers' first impressions of the capabilities of the environment. As they sat before the screen with the menubar visible, (see Figure 8) we asked, "Without pulling them open, tell me what you think each of the menus contains. Now, one by one, pull down the menus and tell me what you think each command is used for. What do the groupings of options in each menu indicate to you?"

To illustrate a portion of this task, Figure 9 shows the File, Project, and Compile Menus. One of the issues that interested us was whether programmers would want to have several projects open at the same time. This probing of the menus provided us with an answer. The Project Menu (Figure 9) shows the New, Open, and Close Project options as they would appear when a project is opened and displayed in a window. Most

[3]As the testing progressed, our concerns were confirmed. Programmers who had even a small amount of exposure (in a college class, for example) to Lisp or Smalltalk more quickly understood the new capabilities of the environment.

users commented that since the New and Open options were "grayed" out, that it was likely that only one project could be open at a time. Several users made comments such as "Think C [13] only lets you have one project open, too. I don't like that limitation."

During a later task, we asked the participants to describe what they thought each of the icons in the interface was used for and what each graphic meant. After the test, we had the test participants fill out a short post-test questionnaire that asked for their impressions of the environment. At the end of the test session, we paid each programmer $100 for participating in the test.

What We Found

Each programmer spent about four hours with us exploring this early version of the software and expressing their thoughts. The videotapes from these sessions contain many insights into the way they understood and used the user interface. We have broken the discussion of our results into two different sections. The first section describes the most critical usability problems we found. The second section discusses the problems the programmers had in understanding the capabilities of the environment.

Usability Problems

The test subjects encountered two major usability problems, as well as several minor ones. We will discuss only the major problems, which were discovering how to split and link panes, and interpreting icons.

Splitting and Linking Panes. One of the questions we wanted to answer was whether programmers would be able to learn how to split and link panes by discovery rather than being taught. All eight participants were able to split panes without effort, but they had a much more difficult time discovering how to hot-link panes together. There appeared to be two problems.

First, the "hot link" metaphor was not obvious to the participants. Several did not notice that red arrows indicated a hot link until the test administrator pointed it out. Even after the participants were aware of the relation of the red arrows to hot-linked panes, they could not determine how to "make the arrows red." This was the second problem; the participants had never seen a window control that could be dragged and dropped. One of the participants actually stated what he (incorrectly) believed to be a Macintosh user-interface rule, namely, "you can't pick up or drag a window control." While this programmer may not have been aware of it, the drag and drop function was a new Macintosh behavior that had recently been released by Apple. At the time of testing, most Macintosh applications did not support the behavior, and this was the cause of the confusion. Recently the dragging support was added to the standard system software. This has made operations such as picking up and dragging controls much more common. Thus we expect the severity of this usability problem to diminish with time.

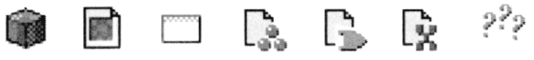


Figure 10. Apple Dylan icons. From left to right: project, source container, comment, class, method, variable, and unsaved sources.

Once the participants discovered (or were shown) how to link panes, they had no trouble understanding and remembering the operation. We concluded that linking panes is initially hard to discover, but makes sense and is remembered after one demonstration. Further experience with this environment confirms that once users know how to link panes, they do not forget.

The development team is exploring several solutions to the initial learning problem with linking panes. They are examining other metaphors and icons for linking, but thus far have not found a new metaphor that is substantially better at representing the linking process. In addition, they are testing the concept of automatic linking. The system has been modified so that when the programmer uses the pane splitter to split a window, the contents of that pane will be automatically linked to its parent pane. The initial reaction to this change has been positive, and seems to address the needs of most programmers.

Interpreting Icons. The second major cluster of usability problems was in the area of interpreting icons. Figure 10 shows the icons that were considered most critical. These icons had been created by the development team to facilitate grouping the code by type. For example, the method icon, which was designed to look like a logic gate superimposed over a document, was intended to be a visual clue that the object was a source record containing a method definition. If programmers picked up on this visual clue, it would move them toward thinking about definitions as individual objects. It would also enable users to quickly scan for different types of definitions within the code since all the methods would have this special "method source record" icon.

README
README

Figure 11. Old and new source container icons. The new icon is based on a folder.

Unfortunately, three usability problems with the icons prevented the programmers from understanding the designer's intention to provide visual clues about the source code structure and definition types. The first problem was the decision to use a document icon (a page of text with the top right-hand corner turned down) for the folder-like source container. While the programmers all understood the document icon from their Macintosh experience, it was not clear to them in what sense groupings of code are like a document. Using the document icon in this way confused the programmers. Although definitions appeared in individual source records, the document icon made them wonder if the source code was stored in files, or in a database, or both. Source containers are more like folders than documents. The designers have subsequently changed the symbol to that of a folder rather than a document (see Figure 11), and have renamed source containers to source folders.

The second problem was that most of the programmers did not understand the meaning of the various source record icons. Different icons were used for comments, definitions, methods, and classes (see Figure 10). The most effective icon was the comment icon, which looked like a yellow Post-it note. Five of the programmers understood it the first time they saw it. The rest saw it as a file card, probably because of its shape. But even the programmers who saw it as a file card understood its relationship to comments. The rest of the icons were less successful at passing the strongest test of an icon, namely, that its meaning be obvious at first glance. Two programmers immediately saw the method icon as a logic gate, and three saw the set symbol as representing classes. The rest of the programmers did not understand the links between the graphics and what they represented. Once the programmers were told what the icons represented, they all saw the link with the type of code. For the remainder of the test, they had no problem linking the icon with the type of code segment. This finding has been confirmed with subsequent users. Users need some aid such as balloon help [2] or a quick reference card to initially understand what the icons mean. Once they understand the relationship between the symbol and the type of source record, they can rely on the visual cue when programming.

Finally, in a finding we anticipated, the 12 x 12 pixel icons with 9-point text were too small to see without moving close to the screen or squinting. In response, the design now allows users to enlarge the icons to 16 x 16 or 32 x 32 pixels, and users can vary the font size up to 36 points.

The most revealing result of this short test was the realization that it is difficult to change the way people do their work, **and that we are only beginning to understand how to structure a user interface that suggests new possibilities.**

Understanding the Capabilities of the Environment

In addition to uncovering usability problems, we wanted to determine the degree to which programmers understood the three conceptually new capabilities of the Apple Dylan environment. These three areas were source code organization, the interactive cross-development model, and the interactive development cycle.

Understanding Source Code Organization. As we discussed earlier, one of the major differences between an object-oriented environment and a file-based environment is the organization of code into a database structure rather than a file structure. The basic unit of code in the environment we tested is a source record rather than a file. The source records include units of code such as methods, classes, variables, and comments. These code fragments are connected through links in a database and user-defined source container groupings rather than through a linear file structure.

Figure 9 shows that the File menu in the prototype had the traditional items for manipulating files, such as New, Open, Close, and Save. When asked what they thought the File menu did, the programmers quickly responded that they expected this to be for manipulating files. When they came to the Project menu, about half of the programmers were confused. They did not understand the differences between the commands on the File and Project menus. If File contains options for managing files, then what do the options in Project do? For instance, one programmer said, *It is not clear whether project stuff should be here (in the Project menu) or in the File menu.*

Some other programmers were not confused, but they did assume that projects were collections of files. For example, a programmer said, *Project is where I manage the files of the project I am working on.*

As the sessions progressed, there were repeated instances in which the programmers referred to files as they discussed other features of the environment. For example, one programmer, in (incorrectly) describing what he thought the arrows on the panes were for said, *Maybe they provide information about what's in the files.*

The expectation that code is organized into files was reinforced by the use of the document icon for source containers and the use of a document in the background of the source record icons. Since the document icon is typically used in Macintosh applications to represent files of text, the programmers assumed that the code for the sample program we created was organized into text files. Clearly, the programmers we tested were (at best) confused about the organization of code in the environment. The familiar looking File and Edit menus and the document icon led the programmers to expect a familiar file structure. In this case, the familiarity worked against the design, leading the programmers to project their past experiences with programming onto this environment.

In response to this confusion, the basic commands for manipulating projects (New, Open, Add to Project, and Close) have been incorporated into the File menu. Informal feedback from users has led the developers to believe that this rethinking of the menu structure, coupled with the new icon for source containers, has eliminated most of the confusion about source code organization.

Understanding the Interactive Cross-Development Model. The dynamic interaction between the development environment and the application under development was not apparent to any of the test subjects, even though they had been briefed on the interactive cross-development model in the pre-test overview material.

When probed about the commands for interacting with the application under development, such as Connect... and Set Runtime... the users did not associate the commands with the cross-development model. In addition, there were several occasions during the testing sessions when the connection to the application was lost, but it was not clear to the users what this meant, or how it affected their test project. We believe at least three factors contributed to the problem, but do not have a solution to the problem.

The first contributory factor was that when the test subjects sat down to begin the test, the development environment was already up and running and connected to an application. No active intervention was needed to connect the two processes.

Secondly, the UI did not provide a strong visual indication of the status of the application process. When the connection was lost, the title of a window changed, and the Set Runtime... and Connect... commands were undimmed in the menus. These subtle changes did not get the users' attention.

And, finally, we believe that the concept of cross-development is fairly new to most mainstream developers. In the months since the test, the UI has been updated to give a more visible indication of the status of the application under development. Unfortunately these changes do not appear to have facilitated the initial shift in thinking that is required when programming with an interactive cross-development model. We would like to run more usability tests to further explore programmers' mental models in this area.

Understanding the Development Cycle. As we discussed earlier, one of the advantages of this environment is that it allows a programmer to compile only the sections of code that have been changed. In the original design of the prototype, the Update command in the Project menu (see Figure 9) was intended to automatically compile only those definitions that had changed. On the other hand, the Recompile command performs a complete recompile of all code in the project, which can take several minutes with a large project. The intention was to keep all of the options that had to do with a project in the Project menu.

The test participants were confused by these options. They could not predict what Update would do. They were split on whether Recompile would compile only changes or the entire program. They were further confused by the fact that there was a Compile menu with a Compile Selection option, which compiled only the section of code that the user had highlighted.

Clearly, these options needed to be reorganized and made more explicit. The new design has eliminated the Compile menu and grouped all the compiling commands on the Project menu. Users can now choose between Compile Selection, Update "Name of Project", and Recompile. When programmers select Recompile, there is a confirmation dialog asking if they want to recompile the whole project.

Conclusion

The development team learned a great deal about the initial prototype's user interface by spending about one week systematically observing and talking with eight programmers. We hope our experience will motivate others to conduct similar usability tests of their own programming projects. We expected to find some usability problems—and we did. In addition, we explored how programmers were thinking about a relatively complex programming environment. We chose to sit and talk with the programmers as they worked. At times we asked questions that probed their understanding of what they were seeing. At other times we watched quietly as they tried to perform tasks. We believe that this method of active intervention worked well for testing a prototype at this early stage of development.

The most revealing result of this short test was the realization that it is difficult to change the way people do their work, and that we are only beginning to understand how to structure a user interface that suggests new possibilities. Inadvertently, the design suggested to the participants that the familiar file-based organization of code had not changed. It also confused users about the possibilities of incremental compilation. Further iterations of testing will tell whether we have solved those problems and whether another sample of programmers will better understand the new capabilities of this environment. We will create online and printed aids for those programmers who will use them, but we are still trying to suggest new possibilities through the organization of the user interface.

Acknowledgments.
We would like to thank Rick Fleischman, Ross Knights, Andrew Shalit, Orca Starbuck, and Carl Waldspurger for their helpful comments.

References

1. ACI, Inc. *Object Master Reference Manual.* ACI, Inc., San Jose, Calif., 1993.
2. Apple Computer, Inc. *Macintosh User's Guide.* Apple Computer, Inc., Cupertino, Calif., 1994.
3. Apple Computer, Inc. Dylan: *An Object-Oriented Dynamic Language.* Apple Computer, Inc., Cupertino, Calif., 1992.
4. Dumas, J., and Redish, J. *A Practical Guide to Usability Testing.* Ablex Publishing Corporation, Norwood, NJ, 1993.
5. Feldman, S.I. Make—A Program for Maintaining Computer Programs. *Software Practice and Experience, volume 9,* (1979), pp. 255–265.
6. Goldberg, *A. Smalltalk-80: The Interactive Programming Environment.* Addison-Wesley, Reading, Mass., 1984.
7. Goldberg, A., and Robson, D. *Smalltalk-80: The Language.* Addison-Wesley, Reading, Mass., 1989.
8. Gould, J.D., and Lewis, C. Designing for usability—key principles and what designers think of them. *Commun. ACM 28,* 3 (Mar. 1985), 300–311.
9. Kernighan, B.W., and Ritchie, D.M. *The C Programming Language, 2d ed.* Prentice-Hall, Englewood Cliffs, NJ, 1988.
10. Microsoft Corporation. *Microsoft Visual Basic Programmer's Guide.* Microsoft Corporation, Redmond, Wash., 1992.
11. Steele, G. *Common Lisp: The Language.* Digital Press, Maynard, Mass., 1984.
12. Stroustrup, B. *The C++ Programming Language,* 2d ed. Addison-Wesley, Reading, Mass., 1991.
13. Symantec Corporation. *Think C for Macintosh User's Guide.* Symantec Corporation, Cupertino, Calif., 1993.
14. Virzi, R. Streamlining the design process: Running fewer subjects. In *Proceedings of the Human Factors Society 34th Annual Meeting,* (1990), 291–294.
15. Virzi, R. Refining the test phase of usability evaluation: How many subjects is enough? *Human Factors 34,* (1992), 457–468.

About the Authors:
JOSEPH DUMAS is a vice president at American Institutes for Research. Current research interests include methods for evaluating the user interface to software. **Author's Present Address:** American Institutes for Research, 70 Westview St. Lexington, MA 02173; email: dumas@forsythe.stanford.edu

PAIGE PARSONS is a human interface specialist at Apple Computer. Current research interests include visual interaction design and browsing metaphors for large information spaces. **Author's Present Address:** Apple Computer, 1 Main Street, 7th Floor, Cambridge, MA 02142. email: parsons@cambridge.apple.com; WWW: http://www.cambridge.apple.com/users/paige.html