Why are Human-Computer Interfaces Difficult to Design and Implement?

Brad A. Myers

July 1993
CMU-CS-93-183

Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Everyone knows that designing and implementing human-computer interfaces is difficult and time-consuming. However, there is little discussion of *why* this is true. Should we expect that a new method is around the corner that will make the design easier? Will the next generation of user interface toolkits make the implementation trivial? No. This article discusses reasons why user interface design and implementation are *inherently* difficult tasks and will remain so for the foreseeable future.

This research was sponsored by the Avionics Lab, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U. S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, Arpa Order No. 7597.

Keywords: User Interface Software, User Interfaces, Human-Computer Interaction, Software Engineering, User Interface Design, User Interface Implementation.

1. Introduction

Most articles about *design* of human-computer interfaces (HCI) start off with a comment like "Because user interfaces are hard to design...." and then propose a method or tool to help. Similarly, articles about user interface *implementation* tools such as toolkits and user interface management systems (UIMSs) will start "Because user interfaces are hard to implement...." But *why* are human-computer interfaces so hard to design and implement, and can we expect this problem to be solved? Like software in general, there is no ‘‘silver bullet’’ [Brooks 87] to make user interface design and implementation easier. In addition to the difficulties associated with *designing* any complex software system, user interfaces add the problems that:

- Designers have difficulty learning the user’s tasks,
- The tasks and domains are complex,
- There are many different aspects to the design which must all be balanced, such as standards, graphic design, technical writing, internationalization, performance, multiple levels of detail, social factors, legal issues, and implementation time,
- The existing theories and guidelines are not sufficient, and
- Iterative design is difficult.

User interfaces are especially hard to *implement* because:

- They are hard to design, requiring iterative implementation,
- They are reactive and must be programmed from the ‘‘inside-out,’’
- They inherently require multiprocessing,
- There are real-time requirements for handling input events,
- The software must be especially robust while supporting aborting and undoing of all actions,
- It is difficult to test user interface software,
- Today’s languages do not provide support for user interfaces,
- The tools to help with user interfaces are extremely complex, and
- Programmers report an added difficulty of modularization of user interface software.

This paper discusses these issues in detail, but first, we summarize why a focus on the user interface is important.

2. Why User Interfaces Are Important

The primary growth area for computers is their use in consumer electronics. This is why computer manufacturers like Apple are getting into the ‘‘personal digital assistant’’ market. The Friend21 project in Japan[1] believes that in the 21st century *everyone* will be using computers for their *everyday* activities [Nonogaki 91]. For the users of these devices, ease-of-use has become a prime factor in decisions about which ones to buy. Time is valuable, people do not want to read manuals, and they want to spend their time accomplishing their goals, not learning how to operate a computer-based system. Usability is now also critical for commercial desktop software. User’s demands on software have changed; they expect to be able to sit down and use software with little or no frustration. Thus, usability is a do or die decision for developers, and is being cited with increasing frequency and explicitness in product advertisements.

Although American industry has invested heavily in information technology, the expected productivity improvements have not been realized [Attewell 93]. Usability at the individual, group and firm level has been cited as a culprit in this productivity paradox. For instance, the ever-changing computer environments caused by new product introductions and upgrades make continual learning demands on workers [Attewell 93].

There is substantial empirical evidence that attention to usability dramatically decreases costs and increases productivity. A model of human performance, and a corroborating empirical study, predicted that a new workstation for telephone operators would decrease productivity despite improved hardware and software. The resulting decision not to buy the new workstation is credited with saving NYNEX an estimated $2 million a year [Gray 92]. A different study reported savings from usability engineering of $41,700 in a small application used by 23,000 marketing personnel, and $6,800,000 for a large business application used by 240,000 employees [Karat 90]. This was attributed to decreased task time, fewer errors, greatly reduced user disruption, reduced burden on support staff, elimination of training, and avoiding changes in software after release. One analysis estimates the mean benefit for finding *each* usability problem at $19,300 [Mantei 88]. A mathematical model based on 11 studies suggests that using

---

[1]Friend21 is a 6-year project started in 1988 with the goal of promoting research and development into next-generation user interfaces, primarily intelligent agents and adaptive interfaces. It is funded at about US$120 million, and is a consortium of 14 major Japanese companies organized by the Ministry of International Trade and Industry. Friend21 stands for Future Personalized Information Environment Development [Nonogaki 91].

software which has undergone thorough usability engineering will save a small project $39,000, a medium project $613,000 and a large project $8,200,000 [Nielsen 93a]. By estimating all the costs associated with usability engineering, another study found that the benefits can be up to 5000 times the cost [Nielsen 93b].

Other studies have shown that it is important to have HCI specialists involved in design. A formal experiment reported that professional HCI designers created interfaces that had fewer errors and supported faster user execution than interfaces designed by programmers [Bailey 93]. One reason is that training and experience in HCI design has a clear impact on the designer's mental model of interfaces and of the user interface design task [Gillan 90]. This implies that HCI design is not simply a matter of luck or common sense, and that experience *using* a computer is not sufficient for *designing* a good user interface, but that specific training in HCI is required.

The importance of a focus on human-computer interaction has been recognized by industry, academia and governments. The Committee to Access the Scope and Direction of Computer Science and Technology of the National Research Council in their report *Computing the Future* lists User Interfaces as one of the six ‘‘core subfields’’ of CS, and notes that it is ‘‘very important’’ or ‘‘central’’ to a number of important application areas such as global change research, computational biology, commercial computing, and electronic libraries [Hartmanis 92]. Two surveys of Information Services practitioners and managers listed Human Interface technologies as the most critical area for organizational impact [Grover 93]. New regulations, such as Directive 90/270 from the Council of European Communities, are being passed that require interfaces to be ‘‘easy to use and adaptable to the operator’’ [Billingsley 93]. ACM has started two new publications about HCI: *Transactions on Computer-Human Interaction* and the magazine *Interactions*. ARPA and NSF in the United States, ESPRIT in Europe and MITI in Japan have all initiated significant HCI initiatives.

3. Why User Interfaces Are Hard to Design

Although the benefits of usability engineering are clear, no-one believes that this solves the problem of making interfaces easy to use. However, there is surprisingly little attention to *why* user interfaces are difficult to design. This section discusses some of the reasons.

3.1 The Difficulty in Knowing Tasks and Users

The first command to user interface designers is ‘‘know thy user.’’ This has been formalized to some extent by the HCI sub-field of ‘‘task analysis.’’ Unfortunately, this is extremely difficult in practice.

Surveys of software in general show that the deep application-specific knowledge required to successfully build large, complex systems is held by only a few developers, and is hard to acquire [Curtis 88].

Furthermore, Don Norman reports:

> My experience is that the ... initial specifications ... are usually wrong, ambiguous or incomplete. In part, this is because they are developed by people who do not understand the real problems faced by the eventual users.... Worse, the users may not know what they want, so having them on the design team is not a solution. Actually, developing correct specifications may not be solvable, because ... a true understanding of a tool can only come through usage, in part because new tools change the system, thereby changing both needs and requirements... All the formalization in the world will not help us solve this problem [Norman 93].

The user interface portion of the code requires even deeper understanding of the users than the design of the functionality since the interface must match the skills, expectations and needs of the intended users. Users are extremely diverse, and the ‘‘individual differences’’ sub-field of HCI is devoted to studying this problem. There is ample evidence that programmers have a difficult time thinking like end-users [Gillan 90]. HCI specialists seem to be better at this, which is one reason their interface designs are easier to use. But finding HCI specialists who are also domain experts is often difficult.

3.2 The Inherent Complexity of Tasks and Applications

An ordinary telephone is pretty easy to use, but modern business phones that can hold, transfer, record, and playback calls can be quite challenging due to the increased complexity. Similarly, Microsoft Word for the Macintosh has about 300 commands and CAD programs like AutoCAD have over 1000. It is clearly impossible for applications with that many functions to have an interface that is as easy to learn and use as one that has only a few functions.

This increased complexity comes from many sources. Partly, it results from the complex requirements in the domain itself. For example, CAD programs must provide techniques for carefully aligning objects, which is not necessary in simple drawing packages. Additional complexity arises from providing a single, generic application that must work for a variety of users and domains. Thus, Microsoft Word has dozens of ways to move the cursor, so that individuals' preferences can be accommodated. CAD programs might provide a dozen different ways to draw a circle so that users can choose the appropriate method for their tasks.

One way to try to overcome complexity is to use metaphors that exploit the users' prior knowledge by making interface objects seem like objects that the user is familiar with. However:

> instead of reducing the absolute complexity of an interface, this approach seeks to increase the familiarity of the concepts.... [However] the inevitable mismatches of the metaphor and its target are a source of new complexities for users. [Carroll 88]

3.3 The Variety of Different Aspects and Requirements

All design involves tradeoffs, but it seems that user interface design involves a much larger number of concerns, and they are the purview of widely different disciplines. User interface design includes considerations about:

1. Standards: An interface will usually need to adhere to standard user interface guidelines, such as the Macintosh, Windows or Motif user interface styles. However, these style guides are usually hard to interpret and apply. Furthermore, the standards will only cover a small part of the user interface design, and will not insure that even this part has high usability. Other ‘‘standards’’ with which a design might need to be compatible include previous versions of the product, and related products from competitors.
2. Graphic Design: An important part of the user interface design is the graphical presentation, including the layout, colors, icon design, and text fonts. This is typically the province of professional graphic designers.
3. Documentation, Messages and Help Text: One study showed that rewriting the help messages, prompts, and documentation to increase their quality had significantly more impact on the usability of a system than varying the interface style [Borenstein 85]. Thus it is important to have good technical writers participating in the design.
4. Internationalization: Many products today will be used by people who speak different languages. Internationalizing an interface is much more difficult than simply translating the text strings, and may include different number, date, and time formats, new input methods, redesigned layouts, different color schemes, and new icons [Russo 93].
5. Performance: Users will not tolerate interfaces that perform too slowly. For example, it was reported that users did not like early versions of the Xerox Star office workstation because there were delays in the response time, even though the users' overall productivity was much higher. Performance concerns explain why moving windows on the Macintosh shows XORed outlines rather than having the

entire window move as on the NeXT. The designer must always balance what is desirable with what will keep up with the mouse.

6. High-level and low-level details: It is not sufficient to get the overall model correct; each low-level detail must also be perfected. If users do not like the placement of the ‘‘control’’ key on the keyboard, or cannot find a menu item, they will not like the interface. Similarly, even if each low-level detail is perfect, if the overall system model does not make sense, the interface may be unusable.
7. External factors: Many systems fail for political, organizational, and social reasons entirely independent of the design of the software. If users perceive that the software will threaten their jobs or status, they will not like it no matter what the user interface. Designers must be aware of the social context in which their system will be used.
8. Legal issues: One way to get a good design is to copy a design that has proven to be workable and popular. Unfortunately, there are many situations where this is illegal today. Lotus sued PaperBack software for copying its menu structure, and Apple has sued a number of companies for copying its user interface. Designers must be aware of which interface elements can be used and which cannot.
9. Time to program and test: There is always a tradeoff between the time to test and perfect a user interface, and the time to ship the product. The more times an interface is iteratively refined, the better it is likely to be, but then it will be later to reach the marketplace.
10. Others: Interfaces that are aimed at special audiences have additional concerns. For example, software that helps multiple users collaborate (computer-supported cooperative work) have interesting design constraints, such as what does Undo mean when multiple people are using the same software? Advanced input devices, such as pen-based gesture recognition, speech, or DataGloves, also raise many interesting issues.

The implication of these requirements is that all user interface design involves tradeoffs, and it is impossible to optimize all criteria at once. Furthermore, people with quite different skills must be involved with different parts of the design.

3.4 Theories and Guidelines Are Not Sufficient

There are many methodologies, theories and guidelines for how to produce a good user interface (each CHI conference proceedings is likely to have a few). Smith and Mosier have compiled 944 guidelines in a 478 page report [Smith 86]. Although there are a number of reports of successful systems created using various methodologies, evidence suggests that the skill of the designers was the primary contributor to the quality of the interface, rather than the method or theory. In fact, there are important counter-examples to even the most basic guidelines. For instance, most sources put *consistency* at the top of lists of guidelines, but

Grudin discusses many cases where consistency is not appropriate. For example, menu systems might have the default selection be the more recent or most likely selection, but still might not use this rule for questions confirming dangerous operations [Grudin 89]. In addition, some of the guidelines in Smith and Mosier are contradictory.

Whereas early papers in HCI were full of experimental laboratory studies of small issues in user interface design, such as the proper menu organization, you rarely see any of these now because the results have failed to generalize. In fact, Tom Landauer says:

> For the most part, useful theory [from cognitive psychology] is impossible, because the behavior of human-computer systems is chaotic or worse, highly complex, dependent on many unpredictable variables, or just too hard to understand. Where it is possible, the use of theory will be constrained and modest, because the theories will be imprecise, will cover only limited aspects of behavior, ... and will not necessarily generalize. [Landauer 91]

3.5 Difficulty of Doing Iterative Design

Due to all the difficulties described above, all HCI professionals and HCI methodologies recommend *iterative design*, where the interface is prototyped and repeatedly redesigned and tested on actual end users. A recent survey reported that 87% of the development projects used iterative design in some form [Myers 92a]. However, this process is also quite difficult.

One important problem is that the designer's intuition about how to fix an observed problem may be wrong, so the new version of the system may be *worse* than the previous version. Therefore, it is difficult to know when to stop iterating. Furthermore, ''... [experimental] data supports the idea that changes made to improve one usability problem may introduce other usability problems'' [Bailey 93]. The same data also showed that while iterating on a poor design does improve it, iteration never gets it to be as good as an interface that was originally well-designed. Thus iterative design does not obviate the need for good designers.

Another important problem is getting ''real'' users with which to test. ''Too often ... testers have to extrapolate from 'problem' users who bring a set of 'hidden agendas' with them to the test session'' [Ballman 93]. The actual users of a product may be different from the buyers, so it is important not to use the buyers as subjects. Participants in tests are usually self-selected, so they are likely to be more interested, motivated, and capable than the actual end users. Each iteration of the testing should involve different users, so a large number of people might be needed.

Finally, iterative testing can be quite long and expensive. Formal tests may take up to 6 weeks so getting answers back to the design team may be slow. A usability lab may cost between $70,000 and $250,000 in capital costs to set up, plus professional staff. When contracted out to a consulting firm, a single usability test may cost between $10,000 and $60,000, and when performed in house, $3000 to $5000 [Abelow 93]. Nielsen provides a survey of the costs for various techniques [Nielsen 93a], and shows that the benefits still outweigh the costs. Furthermore, there are ‘‘discount’’ usability methods that are often sufficient [Nielsen 90]. Still the costs are considerable, and it can take a long time, which conflicts with the desire to get products out quickly.

4. Why User Interfaces Are Hard to Implement

In addition to being hard to *design*, user interfaces are also hard to *implement*. Many surveys have shown that the user interface portion of the software accounts for over half of the code and development time. For example, one survey reports that over a wide class of program types, machine types and tools used, the percent of the design time, the implementation time, the maintenance time, and the code size devoted to the user interface was about 50% [Myers 92a]. In fact, there are a number of important reasons why user interface software will *inherently* be among the most difficult kinds of software to create. For example, if you list the general properties that will make any system difficult to implement, multi-processing, robustness and real-time requirements will be at the top of the list, and these are all present in user interface software.

4.1 Need for Iterative design

The first reason that user interface software is difficult to implement is the need to use iterative design, as discussed above. This means that the conventional software engineering ‘‘waterfall’’ approach to software design, where the user interface is fully specified, then implemented, and later tested, is inadequate. Instead, the specification, implementation, and testing must be intertwined [Swartout 82]. This makes it very difficult to schedule and manage user interface development.

4.2 Reactive Programming

Once the implementation begins, there are a number of properties of user interface software that make it more complex than other kinds of software. One big difference is that modern user interfaces must be written ‘‘inside-out.’’ Rather than structuring the code so that the application is in control, as is usually taught in computer science classes, the application must instead be structured as many subroutines which are called by the user interface toolkit when the user does something. Each subroutine will have stringent time constraints so that it will complete before the user is ready to give the next command. Programmers must be trained to write programs in this way, and it appears to be more difficult for programmers to organize and modularize reactive programs [Rosson 87].

4.3 Multiprocessing

A related issue is that in order to be reactive, user interface software usually must be organized into multiple processes. All modern user interface software environments, including most windowing systems, queue ‘‘event’’ records to deliver the keyboard and mouse inputs from the user to the user interface software. Users expect to be able to abort and undo actions (for example, by typing control-C or Command-dot). Also, if a window’s graphics needs to be redrawn by the application, the window system notifies the application by adding a special ‘‘redraw’’ event to the queue. Therefore, the user interface software must be structured so that it can accept input events at all times, even while executing commands. Consequently, any operations that may take a long time, such as printing, searching, global replace, re-paginating a document, or even repainting the screen, should be executed in a separate process.[2] Furthermore, the window system itself often runs as a separate process. Another motivation for multiple processes is that the user may be involved in multiple ongoing dialogs with the application, for example, in different windows. These dialogs will each need to retain state about what the user has done, and will also interact with each other.

Therefore, programmers creating user interface software will encounter the well-known problems with multiple processes, including synchronization, maintaining consistency among multiple threads, deadlocks, and race conditions.

---

[2]Alternatively, the long jobs could poll for input events in their inner loop, and then check to see how to handle the input, but this is essentially a way to simulate multiple processing.

4.4 Need for Real-time Programming

In addition to the problems involved with multiprocessing, user interface programmers will also encounter the difficulties of real-time programming. Most graphical, direct manipulation interfaces will have objects that are animated or which move around with the mouse. In order for this to be attractive to users, the objects must be redisplayed between 30 and 60 times per second without uneven pauses. Therefore, the programmer must ensure that any necessary processing to calculate the feedback can be guaranteed to finish in about 16 milliseconds. This might involve using less realistic but faster approximations (such as XORed bounding boxes), and complicated incremental algorithms that compute the output based on a single input which has changed, rather than a simpler recalculation based on all inputs.

4.5 Need for Robustness

Naturally, all software has robustness requirements. However, the software that handles the users' inputs has especially stringent requirements because *all* inputs must be gracefully handled. Whereas a programmer might define the interface to an internal procedure to only work when passed a certain type of value, the user interface must always accept any possible input, and continue to operate. Furthermore, unlike internal routines that might abort to a debugger when an erroneous input is discovered, user interface software must respond with a helpful error message, and allow the user to start over or repair the error and continue.

To make the task even more difficult, user interfaces should allow the user to abort and undo any operation. Therefore, the programmer must implement all actions in a way that will allow them to be aborted while executing and reversed after completion. Special data structures and coding styles are often required to support this.

4.6 Low Testability

A related problem is the difficulty of testing user interface software for correctness. While all complex software is difficult to test, one reason that user interface software is more difficult is that automated testing tools are rarely useful for direct manipulation systems, since they have difficulty providing input and testing the output. For ‘‘regression testing’’ (to see if a new version of the software breaks things that used to work in the previous version), tools for conventional software will supply inputs and test the outputs against the values produced by the previous version. However, in a direct manipulation system, if buttons have moved or new items have been added to menus, a transcript of the input events from the previous version may not invoke the desired operations. Furthermore, the outputs of most operations are changes to the screen, which can be impossible for an automatic program to compare to a saved picture since at least something in each screen is likely to have changed between versions.

4.7 No Language Support

Another reason that programming user interface software is difficult is that the programming languages used today do not contain the appropriate features. For example, no popular computer programming language contains primitives for graphical input and output. Many languages, however, have input-output primitives that will read and write strings; for example, C provides `scanf` and `printf`. Unfortunately, using these procedures produces very bad user interfaces, since the user is required to answer questions in a highly-modal style, and there are no facilities for undo or help. Therefore, the built-in input/output facilities of the languages must be ignored and large external libraries must be used instead.

Furthermore, as discussed above, user interface software is reactive and requires multi-processing. Features to support these are missing from languages. Research into user interface software has identified other language features that can make the creation of user interface software easier. For example, most people agree that user interface software should be ‘‘object-oriented’’ but languages do not seem to provide an appropriate object system: Apple had to invent Object Pascal to implement their MacApp framework, and the implementors of Motif and OpenLook for Unix could not find an acceptable object system so they hacked together an object system into C called xtk. One reason C++ is gaining in popularity is the recognized need for an object-oriented style to support user interface programming, but C++ has no graphics primitives or support for multi-processing or reactive programming. A recent book discusses languages for programming user interfaces at length [Myers 92b].

4.8 Complexity of the Tools

Since the programming languages are not sufficient, a large number of tools have been developed to address the user interface portion of the software. Unfortunately, these tools are notoriously difficult to use. Manuals for the tools often run to many volumes and contain hundreds of procedures. For example, the Macintosh ToolBox manuals now fill six books. Some tools even require the programmer to learn an entirely new special-purpose programming language to create the user interface (e.g., the UIL language for defining screen layouts for Motif). Clearly, enormous training is involved in learning to program user interfaces using these tools. In spite of the size and complexities of the tools, they may still not provide sufficient flexibility to achieve the desired effect. For example, in the Macintosh and Motif toolkits, it is easy to have a keyboard accelerator that will perform the same operation as a menu item, but very difficult to have a keyboard command do the same thing as an on-screen button.

It may also be difficult to use the underlying graphics packages, which allow the rectangles, circles and text to be drawn. Since the human eye is quite sensitive to small differences, the graphic displays must essentially be perfect: a single pixel error in alignment will be visible. Most existing graphics packages provide no help with making the displays attractive.

4.9 Difficulty of Modularization

One of the most important ways to make software easier to create and maintain is to appropriately modularize the different parts. The standard admonition in textbooks is that the user interface portion should be separated from the rest of the software, in part so that the user interface can be easily changed (for iterative design). Unfortunately, programmers find in practice that it is difficult or impossible to separate the user interface and application parts [Rosson 87], and changes to the user interface usually require reprogramming parts of the application also. Furthermore, modern user interface toolkits make this problem *harder* because of the widespread use of ‘‘call-back’’ procedures. Usually, each widget (such as menus, scroll bars, buttons, and string input fields) on the screen requires the programmer to supply at least one application procedure to be called when the user operates it. Each type of widget will have its own calling sequence for its call-back procedures. Since an interface may be composed of thousands of widgets, there are thousands of these procedures, which tightly couples the application with the user interface and creates a maintenance nightmare [Myers 92a].

5. Conclusions

While the design and implementation of all complex software is difficult, user interfaces seem to add significant extra challenges. While we can expect research into user interface design and implementation to continue to provide better theories, methodologies and tools, the problems discussed in this paper are not likely to be solved, and the user interface portion will continue to be difficult to design and implement. Furthermore, as new styles of human-computer interaction evolve, such as speech and gesture recognition, intelligent agents, and 3-D visualization, the amount of effort directed to the design and implementation of the user interface can only increase.

Acknowledgements

Thanks to Brad Vander Zanden, James Landay, Dario Giuse and Bernita Myers who provided useful comments on this paper.

References

[Abelow 93] Daniel Abelow.
Wake Up! You've Entered the Transition Zone.
Computer Language 10(3):41-47, March, 1993.

[Attewell 93] P. Attewell.
Information technology and the productivity paradox.
Organizational Linkages and Productivity.
In Douglas Harris and Paul Goodman,
National Academy of Sciences, 1993.

[Bailey 93] Gregg Bailey.
Iterative Methodology and Designer Training in Human-Computer Interface Design.
In *Human Factors in Computing Systems*. Proceedings INTERCHI'93, Amsterdam, The Netherlands, April, 1993.

[Ballman 93] Don Ballman.
User Involvement in the Design Process: Why, When and How.
In *Human Factors in Computing Systems*. Proceedings INTERCHI'93, Amsterdam, The Netherlands, April, 1993.

[Billingsley 93] Pat Billingsley.
1990 EC Directive May Become Driving Force.
SIGCHI Bulletin 25(1):14-18, January, 1993.

[Borenstein 85] Nathaniel Borenstein.
The Design and Evaluation of On-Line Help Systems.
PhD thesis, Computer Science Department, Carnegie Mellon University, April, 1985.
Technical Report CMU-CS-85-151.

[Brooks 87] Frederick P. Brooks, Jr.
No Silver Bullet; Essence and Accidents of Software Engineering.
IEEEComputer 20(4):10-19, April, 1987.

[Carroll 88] John M. Carroll, Robert L. Mack and Wendy A. Kellogg.
Interface Metaphors and User Interface Design.
Handbook of Human-Computer Interaction.
In M. Helander,
Elsevier Science Publishers B.V. (North Holland), 1988, pages 67-85.

[Curtis 88] Bill Curtis, Herb Krasner, and Neil Iscoe.
A Field Study of the Software Design Process for Large Systems.
Communications of the ACM 31(11):1268-1287, November, 1988.

[Gillan 90] Douglass J. Gillan and Sarah D. Breedin.
Designers' Models of the Human-Computer Interface.
In *Human Factors in Computing Systems*, pages 391-398. Proceedings SIGCHI'90, Seattle, WA, April, 1990.

[Gray 92] Wayne D. Gray, Bonnie E. John, and Michael E. Atwood.
The Precis of Project Ernestine, or An Overview of a Validation of GOMS.
In *Human Factors in Computing Systems*, pages 307-312. Proceedings SIGCHI'92, Monterey, CA, May, 1992.

[Grover 93] Varun Grover and Martin Goslar.
Information Technologies for the 1990s: The Executives' View.
Communications of the ACM 36(3):17-19,102-103, March, 1993.

[Grudin 89] Jonathan Grudin.
The Case Against User Interface Consistency.
Communications of the ACM 32(10):1164-1173, October, 1989.

[Hartmanis 92] Juris Hartmanis, et.al.
Computing the Future.
Communications of the ACM 35(11):30-40, November, 1992.

[Karat 90] Clare-Marie Karat.
Cost-Benefit Analysis of Usability Engineering Techniques.
In *Proceedings of the Human Factors Society 34th Annual Meeting, Volume 2*. Orlando, FLA, October, 1990.

[Landauer 91] Thomas K. Landauer.
Let's Get Real: A Position Paper on the Role of Cognitive Psychology in the Design of Humanly Useful and Useable Systems.
Designing Interaction.
In J.M. Carroll,
Cambridge University Press, 1991, pages 60-74.

[Mantei 88] Marilyn M. Mantei and Toby J. Teorey.
Cost/Benefit Analysis for Incorporating Human Factors in the Software Lifecycle.
Communications of the ACM 31(4):428-439, April, 1988.

[Myers 92a] Brad A. Myers and Mary Beth Rosson.
Survey on User Interface Programming.
In *Human Factors in Computing Systems*, pages 195-202. Proceedings SIGCHI'92, Monterey, CA, May, 1992.

[Myers 92b] Brad A. Myers (editor).
Languages for Developing User Interfaces.
Jones and Bartlett, Boston, MA, 1992.

[Nielsen 90] Jakob Nielsen.
Big Paybacks from 'Discount' Usability Engineering.
IEEE Software 7(3):107-108, May, 1990.

[Nielsen 93a] Jakob Nielsen and Thomas K. Landauer.
A Mathematical Model of the Finding of Usability Problems.
In *Human Factors in Computing Systems*. Proceedings INTERCHI'93, Amsterdam, The Netherlands, April, 1993.

[Nielsen 93b] Jakob Nielsen and Victoria K. Phillips.
Estimating the Relative Usability to Two Interfaces: Heuristic, Formal and Empirical Methods Compared.
In *Human Factors in Computing Systems*. Proceedings INTERCHI'93, Amsterdam, The Netherlands, April, 1993.

[Nonogaki 91] Hajime Nonogaki and Hirotada Ueda.
FRIEND21 Project: A Construction of 21st Century Human Interface.
In *Human Factors in Computing Systems*, pages 407-414. Proceedings SIGCHI'91, New Orleans, LA, April, 1991.

[Norman 93] Don Norman.
Apple Computer, Inc. Electronic bulletin board posting.
1993.

[Rosson 87] Mary Beth Rosson, Suzanne Maass, and Wendy A. Kellogg.
Designing for Designers: An Analysis of Design Practices in the Real World.
In *Human Factors in Computing Systems*, pages 137-142. CHI+GI'87, Toronto, Ont., Canada, April, 1987.

[Russo 93] Patricia Russo and Stephen Boor.
How Fluent is Your Interface? Designing for International Users.
In *Human Factors in Computing Systems*, pages 342-347. Proceedings INTERCHI'93, Amsterdam, The Netherlands, April, 1993.

[Smith 86] Sidney L. Smith and Jane N. Mosier.
Guidelines for Designing User Interface Software.
Technical Report ESD-TR-86-278, MITRE, Bedford, MA, August, 1986.

[Swartout 82] W. Swartout and R. Balzer.
The Inevitable Intertwining of Specification and Implementation.
Communications of the ACM 25(7):438-440, July, 1982.