

# TEACHING USER INTERFACE PROTOTYPING\*

*Charles E. Frank  
Northern Kentucky University  
Department of Mathematics & Computer Science  
Highland Heights, KY 41099  
1-859-572-5320  
frank@nku.edu*

*David Naugler  
Southeast Missouri State University  
Department of Computer Science  
Cape Girardeau, MO 62701  
1-573-651-2787  
dnaugler@semo.edu*

*Michael Traina  
Northern Kentucky University  
Department of Mathematics &  
Computer Science  
Highland Heights, KY 41099  
1-859-572-5333  
trainam1@nku.edu*

## ABSTRACT

User interface prototyping is a proven technique for evaluating design alternatives and for obtaining user feedback. It is heavily employed by software development organizations. However, it is often ignored in computer science education. This paper discusses how interface prototypes are used in industry and what research has shown concerning the effectiveness of such prototyping. It evaluates tools that can be used to construct such prototypes and describes how prototyping can be taught in the computer science curriculum. We describe our experiences in teaching students to build interface prototypes and give recommendations based on these experiences.

## INTRODUCTION

Creating a model before building a product is common in engineering disciplines. Architects create visual renderings and build scale models of houses. This allows architects to show their design to customers and to make changes to meet their customers'

---

\* Copyright © 2005 by the Consortium for Computing Sciences in Colleges. Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the CCSC copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Consortium for Computing Sciences in Colleges. To copy otherwise, or to republish, requires a fee and/or specific permission.

needs and desires. After a house is built, modifications are expensive and often impractical.

Software development organizations build prototypes to evaluate design alternatives and to obtain feedback from users in a cost-effective way. After implementation, changes are expensive and time consuming. As Petzold [9] remarks, an important lesson is "if you screw something up at the beginning, it only gets more screwed up when you later try to patch it". Prototyping permits a product to be adjusted to meet the customers' needs at the beginning of the development process.

For users the interface is the product. If users can not or will not use an interface, it doesn't matter how well the functionality is implemented. Products fail because of poor user interfaces.

Developers, no matter how knowledgeable and well intentioned, are probably the worst judges of the interfaces they design. The users' managers are poor judges of the user interface. Only those who will be the interface's users can adequately judge the interface. It doesn't matter how well they articulate what they want or how many designs they sign off on, it is only by experiencing the interface at some level that they can effectively communicate how suitable the interface is for their needs. User interface prototyping allows designers to test the organization of the interface, its wording, layout, overall appearance, appropriateness of menus, toolbars, buttons, even color choices and fonts. User testing using a concrete scaled down version, even a paper version, of a software product provides better feedback than just talking with users.

McCracken and Wolfe [5] note that "computer scientists often do not fully comprehend the need for user-centered design". From discussion with colleagues from a variety of universities, we believe that computer science students do not receive much experience in user interface prototyping and user testing, probably because many of their teachers do not consider it a necessary.

There are several real benefits from including prototyping in the curriculum. First, it is a useful technique used in commercial software development. Second and even more importantly, it requires students as developers to think about the user and the users needs. Software should be attractive, understandable, and should allow users to accomplish their goals. Usable software is much more apt to succeed in the market place.

## **RELATED WORK**

The terms high- and low-fidelity prototype are often used without precise definition. Virzi, Sokolov and Karis [14] note that "a prototype can vary from the final product along several orthogonal dimensions, including breadth of features, degree of functionality, similarity of interaction, and aesthetic refinement." They conclude, "Although prototype fidelity is difficult to define precisely, a prototype that compromises on one or more of these four dimensions in a way that is obvious to the user is a low-fidelity prototype." Using this definition, most prototypes are low-fidelity. What most people mean by low-fidelity is described in [12] by "Low-fidelity prototyping, as we understand it, is the visualization of design ideas at very early stages of the design process. The result is a prototype which is simple and whose development does not need very much time." This is the definition we will use.

There is limited research available on the effectiveness of low-fidelity prototype and on whether paper prototypes are as effective as computer prototypes. Snyder [13] says the questions that need to be asked are "Do paper prototypes find real problems?" and "Does it find the same problems as testing the real interface?" Virzi, Sokolov and Karis [14] in two experiments compared the usability problems uncovered by low- and high-fidelity prototypes. They concluded that "substantially the same sets of usability problems were found in the low- and high-fidelity conditions." Sefelin, Tscheligi, and Giller [12] compared computer-based and paper-prototypes on two systems. They concluded that "paper- and computer-based low-fidelity prototypes lead to almost the same quantity and quality of critical user statements". However, they found that "subjects prefer computer prototypes."

## **PROTOTYPING IN INDUSTRY**

One of the co-authors worked more than twenty years with several software development organizations. His clients included a leading mattress manufacturer, a vacuum cleaner manufacturer, and a leading steel manufacturer for the automobile industry. He used prototyping to ensure that the developers fully understood that part of the business that will use the software. Prototyping greatly simplified and streamlined this step.

His method was to create a reasonable facsimile of the product in the same media as the finished product. The software used to create this interface was not necessarily the software used in the finished product. The idea was to create the prototype as quickly as possible so that the client would give fast approval. However, a consideration is the duplication of work involved. If the method used to create the interface could also be used in software development, then the developers could recognize a time savings.

It is important that the client signs off on the interface. To do this, the client must have the ability to test-drive the facsimile product, or at least see the product in a real setting, even if it only produces made-up data. The prototypes often included the interface and output, but did not necessarily contain working algorithms. Interfaces were often populated often with "pretend" data, hard-coded in the program or from a small database.

Prototyping allows for iterative changes to the software solution prior to final approval. The product approval process must be clearly spelled out by the developer. A mistake made by many developers leaving the prototype process so loosely defined and open to interpretation that the client can make changes seemingly forever.

## **PROTOTYPING IN THE CURRICULUM**

There are two primary places where interface prototyping can naturally be placed in the computer science curriculum. One is in an elective course in human computer interaction (HCI). The other is in a software engineering course, which is often required and may be the capstone course in the degree program. A software engineering course will likely discuss the use of prototypes [11] not only to gain user feedback but also as a proof-of-concept to test feasibility and to test performance. In a smaller way user

testing and interface prototyping and other HCI topics can be minor parts of many computer science courses.

In human computer interaction textbooks, user testing with prototypes is a major focus. McCracken & Wolfe [5] has a chapter on understanding users plus two chapters on prototyping and their use in user testing. Preece [10] has chapters on prototyping and user testing. In *Computing Curricula 2001* [1] (CC2001), user testing and prototyping appear in two elective modules in the human-computer interaction body of knowledge. Human-centered software evaluation (HC3) covers evaluation of software by usability testing. Human-centered software development (HC4) covers prototyping techniques and tools.

While prototyping and its use in user testing are central to human computer interaction textbooks, software engineering textbooks give minor coverage to this topic. This is somewhat understandable since there is a lot of material to cover in a software engineering course. Schach [11] devotes a few pages to rapid prototyping as means of determining the real needs of clients. Client feedback from interacting with the prototype is used to construct specifications. In CC 2001 [1], prototyping appears in the software engineering body of knowledge in software requirements and specification (SE5). SE5 is in the core with a minimum of 4 hours of coverage.

Software engineering courses often involve a semester long team project, which may be assigned by the instructor or may involve real customers. The challenge of completing a large scale design, coding, debugging, and documentation effort leaves little time for building a prototype and assessing user needs.

## **PROTOTYPING TOOLS**

Paper prototyping in particular is an easy to learn technique that works. However, students in computer science and software engineering often respond to it as "fun" but somehow "not professional". A common remark is that it would have been easier and more appropriate to implement an interface in code.

The students we work with are largely computer science majors for whom programming is a core concept and activity. Although they certainly learn theory and take courses that do not directly involve much programming, these courses inform the design and construction of programs. Naturally, our students want to implement since they feel that implementation is an essential part of what they do, an attitude that conflicts with the content of some of the courses they must take.

With a class of computer science students the issue is not really whether or not to implement, but when and what to implement. We would definitely recommend creating and testing a paper prototype as a first step. This forces students to focus on the interface rather than on programming.

There are several approaches to software prototyping but the most important consideration is how large a part of the course you want it to be and what background your students have. The basic practical choices are to use web pages, a drag and drop environment which supports a language with reasonable GUI capabilities or a language with strong and relatively simple GUI support.

Since most students in the computing sciences are familiar with HTML, using web pages for implementing a prototype interface is the most widely available choice. xhtml and CSS (cascading style sheets) provide a limited set of constructs that encourages separation of content and formatting, encourages consistency in appearance, and allows easy overall appearance changes. Effective web interfaces can be implemented with xhtml and CSS. The limitations encourage the focus to remain on the interface. Forms allow the testing of the interface and the judicious use of JavaScript or other scripting alternatives allows the implementation of more interactive pages. Overall, xhtml and CSS work well, especially for web interfaces, if only a working prototype interface with perhaps some simulated functionality is required.

If the prototype interface implementation is part of an overall web-based project, the above approach is unlikely to be adequate. Java Server Pages and servlets, or ASP.NET with VB.NET or C# may be appropriate tools especially if the class is already familiar with these technologies. For Java there are many reasonable drag and drop environments that could be used, and for C# and VB.NET there is Visual Studio.NET. With these, the interface can be reasonably disjoint from the rest of the functionality. If the students are already adequately familiar with one of these, it is an obvious choice. It is, however, very easy to underestimate the time and effort required to teach a language and or an environment and the time and effort required for students to achieve an adequate facility with them. Complex environments take time to reasonably master and modern languages and environments make it easy for students to get lost in what they can do with a tool, rather than what they should do. Overall, these technologies can be effective if the students already have a sufficient background.

If a non-web-based prototype is to be implemented the choices depend on the particular operating system used. A more "native" Graphical User Interface (GUI) will usually be required. Windows programming [9] is complex and tedious and therefore is out of the question. Visual Basic 6.0, old in technology years, and increasingly obsolete provides an intertwined language and drag and drop environment which allow a reasonable Windows' style GUI to be developed. Both Java and C#/Visual Basic.NET have very flexible libraries for GUI programming but both require reasonably knowledgeable programmers to implement a GUI, or some facility with the language and a drag and drop environment with adequate choices.

There are many, sometimes cross platform, choices that might be appropriate in special situations including everything from Smalltalk to Visual Prolog to Icon/Unicon. In spite of their many merits, none of these are sufficiently main stream to be likely choices and all have steep learning curves.

One tempting choice is tcl/tk. Tcl/tk it is freely available, very capable, and available for many platforms. For Python enthusiasts, tkinter is simply tk in Python syntax and it is very easy to convert between tkinter and tk. With tcl/tk, it is deceptively easy to create a simple GUI. Tcl (Tool command language) is a somewhat archaic but capable command language. Tk is a graphics package, written in tcl, for building GUI's. Tk is so popular in the tcl community that it is part of the name and the distribution. One of the goals of tk was to allow GUI interfaces to be provided for command line programs without any modification of the program's code. If the desired functionality (or appropriate stubs) is provided as command line programs, the tk implementation of the interface can be tested

as a complete prototype without students being able to modify the functionality. Any language available for the operating system which allows command line programs can be used to create the functionality. The separation of GUI and functionality makes it easier for students implementing the interface prototype to remain focused in the interface.

Tk is more complex than it seems at first and although good books [2, 15] are available, they are long. Python's tkinter is not yet well documented although [4] is a good start. Mastering tkinter still means learning tk. Before using either tcl/tk or Python for interface implementation, the instructor needs to be quite familiar with tcl/tk or Python and tkinter and be prepared to spend considerable class time.

In summary, the implementation options explored are all appropriate in some circumstances and not in others. The goals of the course, the time available for the interface implementation, and the background of the students are important considerations. In our experience, it is easy to underestimate the time needed. We advise using the simplest option that is appropriate for the course.

## **RESOURCES**

Listening to lectures about user testing using paper prototyping or reading about it in textbooks is not as meaningful to students as witnessing it live. The Nielsen Norman Group has produced a DVD [7] showing construction materials and the mechanics of carrying out a paper prototype. The DVD shows five different user testing sessions using a paper prototype.

After viewing the video, we divided our students into teams of three and asked them to design and test an online pizza delivery web site using paper prototyping. The teams had to perform user testing using their prototypes on other members of the class. From the video, the teams understood what materials to use to build their paper prototypes and how to test them on users.

Books on prototyping are aimed at professional developers. Snyder [13] has devoted a whole book to paper prototyping. She brings her years of consulting experience in describing how to prepare, conduct, and interpret the results of paper prototyping. This book contains a wealth of practical advice. The book's website [8] has an office supplies page which lists materials used to construct paper prototypes and how to find them at Staples and Office Depot.

Krug [3] gives common sense advice on conducting user tests. He recommends that tests be done cheaply and often with a small number of users. He believes that users do not need to be chosen carefully. Much can be learned about improving interface design and the product on a small budget.

## **STUDENT FEEDBACK**

We had our students teams build their paper prototypes and conduct their user testing during class time. This allowed us to give advice on how to represent user interface items in their prototypes and suggestions on how they could improve the way

they conducted user testing. They had fun carrying out the assignment. On an anonymous feedback questionnaire, they stated that they learned a lot.

Not all student observations were positive. Most of these students have work experience as software developers. One student noted that prototyping does not mesh well with the typical corporate "we need it done yesterday" mentality. Others noted that with today's rapid application development tools, computerized prototypes could be constructed just as quickly and would provide more realistic user tests. One student worried that designs might be easy to build on paper but would be hard to implement. Finally, a student noted that he could not see himself walking into his manager's office and requesting glue sticks, sticky tapes and post-it notes to develop a paper prototype to present to customers.

Jakob Nielsen [6] counters such concerns from the perspective of twenty years of usability engineering experience. He states, "With a paper prototype, you can user test early design ideas at an extremely low cost. Doing so lets you fix usability problems before you waste money implementing something that doesn't work."

The students did appreciate that paper prototypes were easy to create and could be changed quickly. However, one exercise in paper prototyping is inadequate training to develop and perform user testing well.

## CONCLUSION

While user interface prototyping is an important tool in software development, it is often ignored in computer science degree programs. Paper prototyping is easily taught. It provides a low cost way to construct prototypes that are easily changed. It is an effective way to teach students the importance of interface design and how to do user testing. Computerized interface prototypes can be taught by building web pages, drag and drop tools, or by tcl/tk or Python tkinter. Whatever approach is taken, user interface prototyping and user testing are important tools students should learn as preparation for a career in software development.

## REFERENCES

- [1] ACM/IEEE-CS Joint Curriculum Task Force, *Computing Curricula 2001*, IEEE Computer Science Press, 2001.
- [2] Foster-Johnson, E., *Graphical Applications with Tcl/Tk*, M&T Books, Foster City, CA, 1997.
- [3] Krug, S., *Don't Make Me Think*, New Riders, Indianapolis, 2000.
- [4] Lutz, M., *Python Programming*, 2<sup>nd</sup> edition, O'Reilly, Sebastopol, CA, 2001.
- [5] McCracken, D. and Wolfe, R., *User-Centered Website Development*, Pearson Prentice Hall, Upper Saddle River, NJ, 2004.
- [6] Nielsen, J, "Paper Prototyping: Getting User Data Before You Code", <http://www.useit.com/alertbox/20030414.html>

- [7] Nielsen Norman Group, *Paper Prototyping: A How-To Video*, 2003.  
(<http://www.nngroup.com/reports/prototyping>)
- [8] Paper Prototyping: <http://www.paperprototyping.com>
- [9] Petzold, C., *Programming Windows*, Fifth Edition, Microsoft Press, Redmond, WA, 1999.
- [10] Preece, J, Rogers, Y., and Sharp, H., *Interaction Design*, John Wiley & Sons, New York, 2001.
- [11] Schach, S. *Object-Oriented & Classical Software Engineering*, 6<sup>th</sup> edition, McGraw Hill, New York, 2005.
- [12] Sefelin, R., Tscheligi, M., and Giller, V., "Paper Prototyping - What is it good for?: A Comparison of Paper- and Computer-based Low-fidelity Prototyping", *Conference on Human Factors in Computing Systems CHI'03*, 778-779, 2003.
- [13] Snyder, C., *Paper Prototyping*, Morgan Kaufmann, San Francisco, 2003.
- [14] Virzi, R., Sokolov, J, and Karis, D., "Usability problem identification using both low- and high-fidelity prototypes", *Proceedings of the SIGCHI conference on Human factions in computing systems* 236-243, 1996.
- [15] Welch, B., *Practical Programming in Tcl and Tk*, 4<sup>th</sup> edition, Prentice Hall PTR, Upper Saddle River, NJ, 2003.