User Interface Specification

Daniel M. Berry

User Interfaces a How Issue?

We used to think that specifying the user interface (UIs) for a system is a How issue rather than a What issue.

That is, it should not be specified in the requirements specification and should be left to the implementers to decide.

However...

However, we have had enough catastrophes in which the culprit was a poor UI on the controlling application that left the operator confused as to what was happening and

he or she made a poor choice of what to do.

or

he or she made the wrong request.

We Know Better Now

We have learned that UIs must be considered at requirements time.

The UIs must be carefully designed along with the functional requirements to be consistent with the functional requirements.

Need to Validate UIs

Many times, it is necessary to validate proposed UIs with usability testing with real, alive users.

Finally, the final UIs must be specified in the requirements specification along with all the functional and nonfunctional requirements.

Knew This All Along

Actually, we knew this all along, because ease-of-use is often a nonfunctional requirement, and is thus a requirement that must be specified.

The Harsh Realities

Moreover, we have learned that if U-I issues are not decided upon and specified in the requirements, it often ends up that it is impossible to add them later to the code that results because the proper hooks have not be left in the code.

Even worse than that, it is often necessary to program the function into the UI framework rather than the other way around.

How to Specify User Interfaces

A cool way to specify the UI of a system is to attach screen diagrams to scenario steps.

Doing so has the effect of showing:

- **when a particular screen or window appears,**
- **how the particular screen or window appears, and**
- **what the system does in response to a particular input, including that of selecting or clicking a particular widget.**

To Examples

Let us now specify a reasonable WIMP UI for the *Sensus* system which we have used as an example before.

Recall...

But Whoa!

UCs & Ss are not specifications!

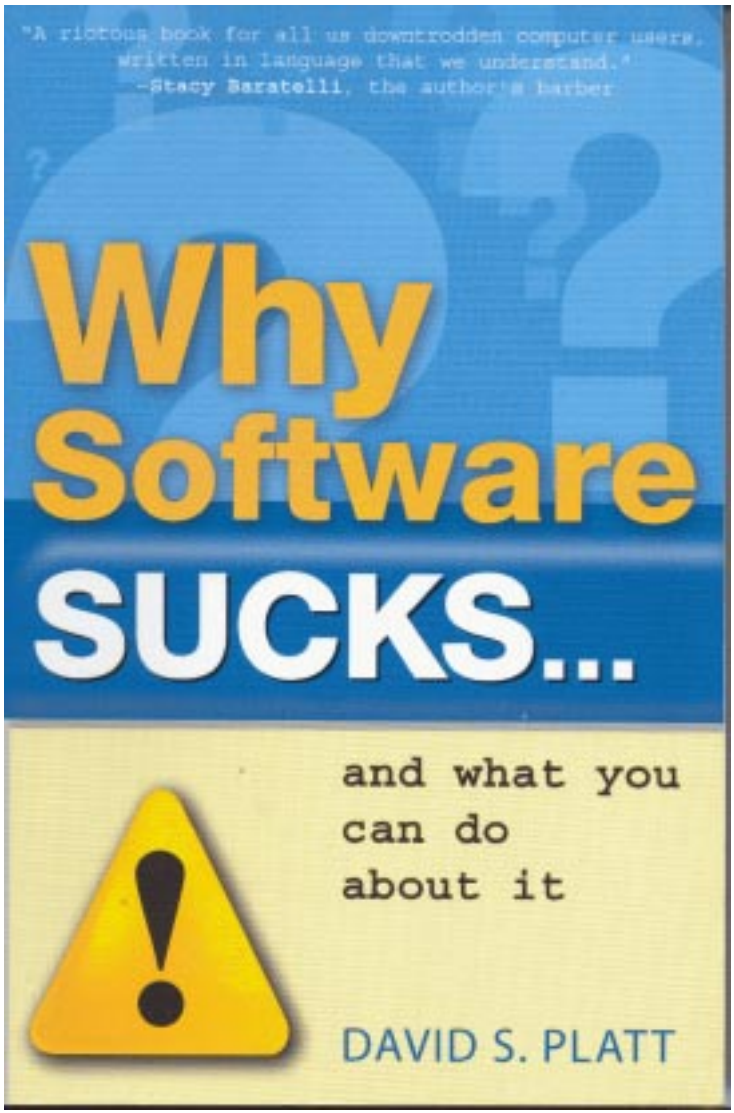
They only *illustrate* functionality from the user's point of view.

Thus a UI specification tied to UCs & Ss is not really a specification.

Tie UI Specs to Specs

For each scenario step with a numbered marker, find the states or transitions in your state machine or process diagrams that correspond to the scenario step, and give the same numbered marker to these states or transitions.

You may need to indicate which transition is taken in response to any textual input or to any widget selection or clicking.


"A riotous book for all us downtrodden computer users, written in language that we understand."
-Stacy Baratelli, the author's barber
Why Software SUCKS...
and what you can do about it
DAVID S. PLATT

Why Software Sucks

In a nutshell,

Software sucks because UIs suck!

For many a program, its UI is not obvious.

The typical non-expert user cannot figure out to get the program to do what he or she wants it to do.

No or Poor Manual

There is often no manual.

When there is a manual, also it sucks, because one cannot easily find the answer to his or her questions about how to use the program.

Poor Help System

There is often a help system, …

but its index is not very helpful in finding answers to specific questions.

User Feels Stupid

Often, a user is left feeling stupid by his or her inability to get the program to do what he or she wants it to do.

The reality is that the user is fine, but the UI is stupid.

The Way it Ought to Be

A program should be designed in a way that makes consulting a manual or help system unnecessary.

Its UI should guide the user through solutions to his or her problems.

The Way it Usually Is

Instead, the user finds that he or she has to understand the inner workings of the program to use it.

Why Do UIs Suck?

David Platt suggests that the reason a program's UI sucks is that the program's programmers, not professional UI designers, design and implement the program's UI.

The typical programmer programs the interface for a user like him or herself.

The Typical Programmer

The typical programmer wants to be *in control* of all options.

Programming is the *ultimate* expression of this control!

The Typical User

The typical user wants the program to do only what he or she wants; i.e., he or she could not give a s--t about all the options.

How a Program Should Behave

The program should do normally what most people want as a default *without* asking the user to make choices that are probably unintelligible, e.g.,

‘‘Allow ❑ / Disallow ❑ cookies.’’

The program should have an optionally invoked “Preferences” section that guides the user in making his or her choices intelligently, possibly explaining the implications of each choice.

Platt's Law of UI Design

David Platt's First, Last, and Only Law of UI Design:

Know Thy User, for He Is Not Thee.

Control vs. Ease of Use

You, the programmer, may want control.

The typical user wants ease of use.

Example

Directory Assistance:

Control: AT&T's directory assistance just tells you the phone number. You have to dial if you want to.

Ease of use: Verizon's directory assistance tells you the phone number and then dials it for you.

Which choice reflects the way most users operate?

I Don't Care

The typical user says, "I don't care how your program works!" (Actually he or she probably says it with slightly different words!)

Many a programmer forces the user to understand how the program the programmer wrote works in order to use it properly.

Example

The question that the typical text editor or word processor asks when you try to exit the program is, “The text in the file *F* has been changed. Do you want to save the changes?”

Implication of the Question

This question forces the user to understand that the way the program works is that it first reads the file contents into the memory, it modifies the in-memory copy, and then at the end, for the changes to be permanent, the current in-memory must be written back to the file.

A better question is “Do you want to throw away every change you have just done?”

Bad vs Good Feature

On a Windows system, when you select a file *F* and then press the "Delete" key, …

unless you have figured out how to disable what is about to happen, you get a dialog box that asks, "Are you sure that you want to send *F* to the Recycle Bin?"

Do You Really Want to Be Asked?

When you turn a car's ignition on or off, does the car ask you if you really want to do what you have just done?

The purpose of the recycle bin is to allow recoverable deletes. So why is it necessary to ask if the user is sure that he or she wants a file sent to the recycle bin?

A Better Idea

A better idea is to make as many operations as possible undoable and redoable.

The possible exception would be emptying the recycle bin.

Even that can be made undoable by tying the emptying operation with ???

Lostness Formula

Tom Tullis and Bill Albert(2008) have offered a formula for calculating how lost the user of a Web site is:

R **= the minimum number of pages that must be visited to do the task at hand.**

N **= the number of different pages actually visited while doing the task.**

S **= the number of pages actually visited while doing the task, including revisits.**

Lostness Formula, Cont'd

$$L = \text{sqrt}\left(\left(\frac{N}{S} - 1\right)^2 + \left(\frac{R}{N} - 1\right)^2\right)$$

L = lostness, [0 .. 1],

0 = not lost at all.

1 = totally lost.

0.4 is already bad.

My Advice

KIS

Keep it Simple!