

developer.***A Web Magazine for Software Developers**

Those “Minor” Usability Annoyances

By Daniel Read

“Most software is used in a business context, so most victims of bad interaction are paid for their suffering. Their job forces them to use software, so they cannot choose not to use it—they can only tolerate it as well as they can.” — Alan Cooper, The Inmates Are Running the Asylum (Page 34)

As I write this, I am contracted to a relatively new team tasked with totally rebuilding a fairly complex and sizable mission-critical application—an internal business application, used by a couple hundred users located around the world. Development on the original version started over three years ago. The application has been under continuous development since that time, and in production about two years. I’ve been the technical lead and architect for the past two years; I joined the team shortly before the application went into production, and after it had been in development for about a year. So I am quite familiar with the original version, well aware of its good and bad points.

The reasons the business decided to rebuild the application are fairly typical: the original architecture was short-sighted and does not scale; the code base is convoluted and fragile, and therefore costly to maintain and extend; and the technology and platform are out of line with the corporation’s recently solidified standard technology stack. Given the importance of the application, and given the requirements for critical new functionality that the current architecture won’t support, deciding to rebuild from scratch was a no-brainer.

Oh yes, there’s one other thing: a significant percentage of the application’s users absolutely hate it. One user even prefers to refer to himself as a “victim” of the system rather than as a user. Though, from a giant corporation’s point of view, this fact alone would not have been enough to warrant a total rebuild, it is still a significant fact, and one that has consistently bothered me for two years. I feel for these users.

When I first joined the team as the senior technical person two years ago, I went to a training class for the application during my first week. One hour into the class, I was slouched low in my seat, wondering what I had gotten myself into, as I watched the application crash repeatedly and listened to the trainer apologize for all the strange and awkward quirks. In only an hour, I had filled up two pages with observations of problems I saw with the application. Most of these problems were usability problems. Even today, after many, many of the application’s early flaws have been resolved, I still sometimes literally experience pain while watching users sit in front of the application and try to do their jobs.

Why do the users hate it so? Does it lack the functions they need? Nope. With a few exceptions, the application essentially performs the functions the users need. Does it

crash a lot? It used to, but not anymore. Does it not make available the data they require? Got that covered. Is it slow? It's no speed demon, but it's not a dog either.

The users' dislike of the application is actually caused by a swarm of tiny (and some not-so-tiny) usability annoyances. No single annoyance is enough to cause a big problem, but there are so many, they gang up on you like mosquitoes flying around your head. Beyond the annoyance factor, these usability problems are also a significant productivity drain. Switching metaphors, these small problems are like barnacles gathered on the bottom of a boat: no single barnacle is going to slow the boat down measurably, but let enough of them implant themselves, and a significant drag will result.

The intensity of the user's stored-up ire truly became clear to me last week. The occasion was a prototype review meeting. In the process of rebuilding the application, the new team is naturally redesigning the entire user interface. We had already spent several weeks gathering requirements and documenting them in use cases. The use cases were approved by the subject matter experts and the users, and the development team had developed some initial user interface prototypes. The point of this meeting was to get the prototype in front of the users in order to get their feedback and make sure we were moving in the right direction. The meeting was a success. The users liked the direction we are going with the new interface.

What was remarkable, however, was that throughout the meeting the users often had a hard time concentrating on the prototype in front of them because they would get caught up ranting about all the things they hated about the current version. One person would notice something that we had done well in the prototype and then relate it back to something she hated about the current application, which would immediately prompt three other people to offer their own horror stories. The complaining was mostly good natured—no one was really angry or upset—but their frustration was plain and undisguised.

This experience magnified for me two years of supporting this application and brought three conclusions to mind:

First, three years ago, the original team, while they did many things well, failed to design and build a user interface that offered a considerate, congruous user experience that anticipates the user's needs. Furthermore, the user interface code they developed was unstable, difficult to maintain, and sloppily constructed, both on the inside and the outside. These mistakes cost the project dearly in terms of good will and acceptance from the user community.

Second, the current team, in place for the last two years (which has only one developer—luckily the best one—from the original team), could have done more to overhaul the user interface and get rid of many, if not most, of the problems.

And third, since the current team never performed any kind of overhaul or cleanup, we consequently never established a standard that could be consistently applied to the large

amount of functionality that we've added during the last two years, which further contributed to the inconsistencies and annoyances.

While I am free of blame on the first point, I accept full responsibility for the second and the third.

(To be fair, the original team was up against some difficult circumstances, and, as I said, did many things well. Also, the current team boasts a long list of tremendous accomplishments, having rescued the application from impending failure and having added a huge amount of new functionality that has provided real value to the business. And many usability gains have been made. Furthermore, the business management of the development team has always been under heavy pressure to continually add new functionality and change existing functionality to support a fast growing business, and was never given any mandate to use its tight development resources for a usability overhaul. My three conclusions are more in the spirit of an academic "What can we learn from this?" exercise. Of course, at the end of any enterprise, it's always easy to look back and say "We could have done more; we could have done it better.")

The irony is that this application I'm talking about is pretty damned decent, and in many ways I'm proud to be associated with it. It's fairly stable; it performs its calculations accurately; it meets the majority of the myriad fast changing needs of the business, and has I dare say provided them a real edge in the marketplace; it provides an impressive list of features; it offers an impressive array of reports; it has a decent database; it integrates well with other applications in the enterprise...I could go on extolling its positive qualities.

What makes the situation ironic is that, no matter how many positive qualities I could list, no matter how convincingly I could make an argument that, given the failure rate of development projects this size, this one is quite successful, many users still have a visceral dislike of the application. The mosquitoes still swarm around their heads—the barnacles are still slowing down the boat—and that's what the users naturally focus on.

By way of example, let me list some of the usability annoyances and problems this application has (or has had at one time). The application in question is a Windows application, so some of the usability comments below will be specific to Windows applications.

- One of the biggest problems is that many facilities of the interface force the users to do things one step at a time rather than providing shortcuts or ways to effect aggregate changes. A function that, with a little creativity in the design, could be performed with just a few mouse clicks instead takes dozens of mouse clicks. The user is forced to iterate through a series of repetitive steps. Imagine a spreadsheet without Copy-Paste or "Fill Down" functions, and you'll get the idea of what I mean. My opinion on what led to these flaws is that the original developers translated the structure of the data too literally into the user interface. Just because the data is best stored in a certain way, that does not necessarily mean that the users should be burdened with interacting with it that way.

- Another pervasive problem that further exacerbates the forced repetitive operations problem is that there are pop-up prompts and informational dialog boxes everywhere. These pop-ups are usually variations on three themes: Before performing an operation, displaying a confirmation prompt such as, *“Are you sure you want to do that?”*; and after performing an operation, displaying a message such as: *“Function performed successfully.”* or *“Now that we performed the function, just to be safe this message will inform you of some ramification of having performed it.”* Each pop-up forces the user to either answer a question or simply press the “OK” button. These dialogs do what Gerald Weinberg calls “inflicting help” on the user. This plague of pop-ups results from two things: first, good intentions on the part of the developers and designers, and second, inherent flaws in the user interface design that leave traps for the user to fall into; rather than removing the traps, we’ve just tried to mark them clearly with warnings. Some amount of these kinds of pop-ups might be necessary, but in our application their use is gratuitous.
- Throughout the application, error messages and prompts are cryptic. Imagine sitting for several minutes waiting for a complex simulation process to run, only to have it fail, and the failure message you get is “Element not found.” Similarly, pop-ups that ask the user to choose one course of action or another do not adequately explain what is really at stake, and the user is forced to guess and hope for the best.
- The application offers a wide variety of reports, and the users depend heavily on them. However, the launching of reports is not context sensitive. In other words, if you are sitting on an “Edit Customer” window with the “Ajax Widgets” customer loaded, you’d like to be able to launch a customer report with one mouse click. After all, the application knows what customer you have loaded. Instead, you have to click on the “Reports” menu, navigate down to the “Customer Reports” submenu, find the report you want, click on it; then a parameter input dialog box pops up, presenting you with a list of all the customers in the system; you must scroll down and find the “Ajax Widgets” customer and select it; then you click the “Run Report” button and your report will come up. Taking it one step further, instead of a simple one-level parameter that presents a list of customers, imagine a three or four level hierarchical parameter with cascading drill-down lists. You can imagine that this lack of context sensitivity drives the users crazy.
- At a more granular level the windows themselves have tons of little annoyances: some windows have “OK” and “Cancel” buttons; some have “Save”, “Save and Close”, and “Cancel” buttons; some have “OK”, “Apply”, and “Cancel” buttons; some have “Save” and “Cancel” buttons. These clusters of buttons are sometimes found in the lower left corner of the window, sometimes in the upper right, sometimes in the lower center. Some “OK” buttons have designated “accelerator key” shortcuts (for example, allowing the user to use `Alt-O` to activate the button from the keyboard), some “OK” buttons do not. Some windows allow you to activate the “OK” button with the `Enter` key, some do not. Some windows allow you to activate the “Cancel” button with the `Esc` key, some do not. Many forms have a crazy, undefined “tab order,” meaning that when the user hits the `Tab` key to move from field to field, it jumps all over the window instead of going from field to field in a

logical order. Half the windows have that little “What’s This?” help question mark turned on in the upper right corner, but when you try to use it, you find out that no help text has been defined for any of the window’s elements. Many times, field and list widths are not wide enough so that the values get truncated and you have to scroll to the right to read what’s in the field. Grids on many of the forms require extreme horizontal scrolling. Important operations are hidden in right-mouse-click pop-up menus and do not also appear on the main menus or toolbars. Lists are not ordered in any logical way so the user must hunt through the list to find the correct item. Window elements and controls are placed in proximity to each other in ways that suggest interrelationships that do not exist. Conversely, elements and controls that are in fact interrelated are placed apart in ways that suggest that they are not related.

I could go on and on with these kinds of examples, but I’ll spare you.

So what can software development teams do? Obviously, it would be ideal to not end up in this situation at all. Some tips for avoiding a usability mess in the first place:

- Carefully consider the interaction design of your application. Make it a priority, not an afterthought.
- Assign developers who are experienced in building user interfaces, especially using your chosen technology. *Every* technology/platform/toolset that allows the building of graphical user interfaces is full of quirks, hacks, shortcuts, and bugs, and it really does take someone who has experience with your chosen technology and tools to do a good job and at the same time be productive.
- Hire developers who care about and keep track of the small aesthetic and interaction details, since it is the neglect of these that lead to usability problems. Even a brilliantly conceived interaction metaphor can be ruined by a lack of attention to detail in its construction.
- Perform usability reviews with experienced designers, and of course, with your users.
- Early on, develop and publish a list of interaction design principles. Then develop and publish a standards document for user interface design to ensure consistency. Of course, your standards should reflect your principles, but your principles will guide your developers when the standards fall short.
- Buy copies of the important interaction design books for your developers.
- Listen carefully to your users, especially to what they are *not* saying.
- Be very careful about following too closely the design of your database and/or object model when designing your user interface. Developers think in and work with the abstractions at the OO and DBMS layers, but your users could give a damn about the structure of those abstractions. Don’t let the structure of your database and objects drive the structure and flow of your user interface. Let the requirements of the user drive it.
- If the option is available to you, spend a little money on commercially available user interface widgets. Often, the widgets that come by default with your development tool are lackluster at best or totally deficient at worst.

- If you feel you need to, hire an outside consultant to come in and work with your team on the initial development of interaction metaphors. There are consultants who specialize in this. They can perform usability studies of existing applications or manual processes to help you develop a metaphor that best suits the needs of your users. After some initial consultation, you can bring this person back for periodic reviews. This can pay off big if there is a lot riding on the usability of your application or if the design presents special challenges.
- Take a good look at the cross sections of users you have. You will likely find that different user constituencies have very different needs and goals. Some might be very data entry and keyboard driven. Others might only go into the application sporadically to perform very specific tasks. Others might only need easy access to certain reports. (This is the basic approach of Alan Cooper's Goal-Directed Design methodology.)
- Don't forget to consider users with special needs (such as blindness, poor eyesight, or color blindness) and users from other cultures.
- Finally, be careful not to overcompensate and try to make your application *too* usable. It is of course important to consider brand new users of your application to make sure that it is easy to learn and understand. However, new users are only new users for a short time, and if you cripple them with "helpers," they may grow to hate the application as they become more experienced.

But what can you do if your application is already in a usability mess? First, you have to consider what kind of mess you have. There are at least two kinds of messes: one, your interaction metaphor is all wrong, or two, your interaction metaphor is essentially sound, but you've got a lot of usability annoyances and sloppy construction work that are screwing it up. In the first case, your foundation needs to be replaced (which will likely necessitate the replacement of almost everything else along with it), whereas in the second case, you just need to clean up all the little things. In either case, you've got bad news, but in the latter case, the news is not nearly as bad.

In the former case, you are pretty much starting over, so I point you back to the above list of advice aimed at teams which are starting from scratch. For an application simply in need of cleanup, I offer the following list of advice and techniques for your consideration:

First, techniques for finding out what problems you have:

- Start off by having a frank discussion with your users about usability issues they have. Let them know you are sincere in wanting correct the issues and that you understand their pain. Sometimes a simple apology and a demonstration that you recognize the problem can work wonders. Be sure to meet with actual *users*, not with their managers who don't use the application. Initially start with a list of their top ten desired changes. From there, you can work towards an exhaustive list.
- Send one or more members of your team (preferably the developers who will do the work) to shadow different users. Have them observe the users doing their jobs and using the application and have them take notes on what they see. It might be ideal if they could do this as silently and unobtrusively as possible, but there are also

significant benefits to be had from interacting with the users as they work and asking questions. Identify users of different types so that you can get a good cross section. Also, spend time with both experienced users, who know where all the potholes and quicksand traps are, and with brand new users

- Have someone from your team sit in on some training classes. Listen especially for those things that the trainer feels the need to provide special explanations for or apologies for. Try to spy on the new users and observe what difficulties they have.
- Assign one or more experienced developers to perform a survey of your user interface code looking for usability problems and inconsistencies. Keep a screen-by-screen list of the problems you find.
- If you've got the money to spend, hire an outside usability consultant to come in and perform a usability study. These studies can be quite sophisticated—and revealing. Many consultants have special equipment such as cameras that track eye movements, microphones, and screen recorders.

Second, techniques for fixing the problems you find:

- The most extreme technique is to just stop everything and overhaul the user interface. This is generally only warranted if you have major problems, and only feasible if you can a) get management buy-in, or b) do it in secret. Few projects will be able to indulge in the luxury of an overhaul, even if they really need it, and one of the more gradual techniques described below will have to suffice.
- A close cousin to the complete overhaul is a triage approach, in which you stop new development temporarily and remedy only the most heinous problems. After you have identified a “Top Ten” list of usability problems, you can take a break from new feature development to fix these. Your list does not of course have exactly ten items on it. The idea is to leverage the 80/20 rule: expend 20% of the effort that a complete overhaul would take in order to reap 80% of the benefits. If that means you can only fix five problems, then that's a whole lot better than fixing none. If you can fix twelve, even better. After the initial effort, you can downshift to one of the even *more* gradual techniques described below.
- If you can generate an exhaustive list of usability problems, then you can work these gradually over a longer period of time by slipping the fixes into your regular releases. One way to accomplish this is to append a usability fix to each new feature. For example, if Sue is developing a new search feature for the web site, then she could pick a usability problem from the list and spend a little extra time working on the fix, hopefully expanding the total time to develop the new search feature only slightly. This stealth approach is especially effective when management will not get behind any above-ground effort to improve usability. Over time, quietly, the application will be transformed. Ideally, your exhaustive list of problems would be classified by risk and difficulty, which would aid developers in choosing a task.
- A more visible compromise with a management that has trouble approving a concerted usability improvement effort is “Feature Fix Fridays.” In this approach, the team spends Fridays putting aside their normal work to tackle usability issues. This can easily be scaled back to one Friday per month, or one Friday per release

cycle, or whatever works for your situation. Of course, this is only practical for a lot of smaller work that can be spread out and fixed in a day. Many problems can of course take many days to resolve.

When resolving usability problems, it is especially important to pay attention to consistency across the application. If possible, when fixing an issue, fix it in the same way throughout the application. Few things are more annoying in an application than behavior differences in different parts of the user interface. Even if a user interface sucks, it's at least preferable for it to suck consistently. Not to mention, visually speaking, inconsistencies can make your application look hacked together. Even if the code is total spaghetti behind the scenes, no one will be the wiser if the façade is at least consistent and presentable. ("Pay no attention to the man behind the curtain!")

My hope at this point is that the new team developing this re-write of the application can learn from the previous teams' mistakes and deliver a user interface relatively free of these kinds of problems. At this point, I have no reason to believe that we will fail. The "victims" of the current system have made their expectations clear to us.

###

Daniel Read is editor and publisher of the **developer.*** web magazine. He lives in Atlanta, GA, where he works as a software architect. He is currently at work on a book about software development crafted for a business audience.