How to design a decent User Interface

Take a look at software from a user's point of view and improve your applications

Corinna Habets
pallas@koeln.ccc.de

17. November 2006

1 Motivation

Nowadays computers are everywhere. From the holy halls of mainframes they dropped down into people's mere living rooms. That means that more and more people work with computers who have no idea about their inner workings. **And don't want to have!**
For most people out there a computer is only a tool, helping them to get their work done. Good user interfaces (UI) make their lives easier, bad ones make them harder.

But why are there so many poor interfaces? It's probably not because developers around the world sit down and decide to implement an interface that no one will be able to use. So let's look at some of the reasons that lead to mediocre or bad UIs.

2 Reasons for poor UIs

- **Lack of Awareness**
  There are huge differences in computer-experience and -knowledge between those who program and those who is programmed for. Many things that are obvious to developers are anything but obvious to the average computer user.

  Many developers aren't really aware of this gap. They think they know what users want, because they themselves are users, aren't they? Unfortunately not. At least not users comparable to average users. Furthermore, as soon as you start implementing an application you aquire an insider perspective. From this perspective it is hard to impossible to notice what problems users will have with the UI.

- **Lack of Training**
  This goes hand in hand with the above: Often the one who implements the UI is also the one who has to design it. While she may be a magnificent programmer and can make Qt do anything she wants, that doesn't mean she knows anything about UI design. These are different tasks and you need different qualifications for them.

- **People are fuzzy**
  Computer-people tend to prefer reliable facts. It's easier to focus on technology with its fixed numbers and the answer either being 42 or not.
  Contrary to this, people are fuzzy. They vary greatly in experience and "performance".

The performance may even vary in the same individual from day to day. The answer is not 42 but more like "23 out of 42 people had problems finding the "Search"-button." It's often up to interpretation[1] what might be the best thing to do. (But in our example it's quite clear that something needs to be done with the button. Over 50% is a lot!)

Usability is also about trade-offs, e.g. between making it easy for the novice or making it efficient for the expert user. To take such decisions it is helpful to know exactly who the target users of the application are. Do what is best for them. (More in 5)

- **Too little time / money**
  In private projects as well as in companies there can be too little time and/or money.
- **Pressure from higher instances**
  In a commercial environment, there can be demands which the developer has to obey, even if they worsen the UI. Pressure could come e.g. from management, marketing, or the client.

Some of these reasons are more understandable than others. In any case, why should you walk the extra mile?

3 Why it's worth to do it right

I'm about to reveal a sad truth: An average user will never know about the brilliant algorithm you implemented, how efficient it works or how clean and well maintained your code is.

All the user will ever see of your application is the graphical user interface (GUI). As far as he is concerned, the GUI **is** the application. That's why the judgement of an application depends largely on the UI.

Turn this into your advantage. If you succeed in publishing an application that "gets it right" you stand out from the crowd. After all, you want people to use your software, don't you? If not, what's the use developing anything?

But what's a developer to do to get it right? Meet the users!

4 Users as human beings

There's one thing all users have in common: They are human beings and as such have common abilities and limitations.

4.1 Timing

0.1s Perception of cause and effect

If you don't react to a user action within 100ms, the user will wonder whether his request was received.

Implication: For **everything** that can possibly last longer than 100ms make your UI display a busy cursor (short waiting-time expected) or a progress bar (longer waiting-time expected). If the user never gets to see either of them, fine. If not, you prevent random clicks from a bewildered user who is trying to get a reaction.

Bear in mind that most users have slower systems than the one you are developing on.

[1]If you need numbers to convince a number-centric person, there are also formal measures of usability, e.g. to estimate the time to operate different button arrangements. (None of these methods is discussed here. If you'd like to google, try "Fitt's Law" or "GOMS Keystroke Model" for a start.)

1.0s Minimum reaction time for unexpected events[2]

Implications: If you need a user to react to something within a given timeframe, 1s is the time it takes only to realize that something unexpected happened. It also means, that if you ignore the 100ms-rule (what you should not do!) you have 1s before the random clicking starts.

10.0s Typical attention span

Implication: 10s should be the upper limit for one step of a task, e.g. 1 screen of a dialog.

One might feel tempted to ignore the implications above to save time. And if it's faster on the whole, users will appreciate that, won't they? In fact **they won't**. The **perceived** speed of an application is more important than the acual speed. That's why you should always indicate that your application received a user's command, even when it's not yet finished proccessing it.

It's also good practice to give the control back to the user. Don't block everything just because your application is computing something. If all that users can do, is to stare at the screen and wait, it will seem like forever.

4.2 Memory

Our short term memory is limited to 7 +/- 2 items at the same time

It sounds more than it actually is. People are far better at recognizing things than at recalling them from thin air. In a GUI, even if you have no idea what it's about, you can at least look at the menus and might *get* an idea.
With a command line you can't. It takes a great deal more of learning.

Implications: Recognition is better, i.e. easier than recall. For people being able to recognize all available commands, these commands have to be **visible**.
Here we come to a trade-off: On the one hand everything needs to be in reach of the user, but on the other hand we don't want to confuse users with a screen full of buttons.
Resolve this by making all commands available from the menus. The most frequent commands can additionally be in the toolbar (if there is one) to grant faster access. (Don't guess which are the most frequent - test it.)

Wondering why only the most frequent? Why not throw anything in the toolbar to make command invocation faster? Because it takes people time to decide, which command is the right one. The more choice the longer takes the decision. Cluttering frequent commands with infrequent ones slows users down.

4.3 Attention

Only one task can have conscious attention

If we do two things at a time that need conscious attention, say reading while solving a sudoku, our "performance" suffers. Luckily the brain's ability to learn and thus automate actions is huge. That's why we can think about an algorithm (requires attention) while walking (automated). Many things we do daily, even complex tasks like driving a car, can run automatically. But for problem-solving we need conscious attention: Writing a text, preparing slides for a talk, calculating statistics, ...
Many of these problem-solving tasks involve a computer and at least one application: Text-editor, slide-editor, statistics/spreadsheet software, ...

[2] 1s is also the natural flow of taking turns in (human) conversations.

The task is creating the text, slides, or calculations. The task is **not** using the application per se.

Implications: An application with a decent UI lets the users automate its operation as quickly as possible. Only then can users forget about the application and concentrate on the real work. This has a lot to do with consistency of the UI, i.e. that the reactions of the application are predictable to the user. The concept of consistency of a UI is broad and rather blurry. Let's pick one part that is easier to tackle and a great improvement when done right: **Text consistency**.

1. **Avoid ambiguity**
   Language is ambigious. Take "cock" for example. Of course I am talking of the animal. What were you thinking of? ;)
   In everyday life there is lots of context that lets us figure out what is meant. With isolated text this is often impossible. Commands in an application are mostly isolated text. Let's say you find the command "View Settings" in a 3D-rendering-software where you can configure multiple camera views. Now tell me, will invoking this command show the settings of a certain view or show some other settings? The worst case would be that the command existed in different places with both outcomes.

2. **Always use the same word when refering to the same thing**
   When my father got the information for his internet connection, he also received a "passphrase" for logging in into the providers configuration pages. We needed it to configure his email account. The login screen wanted a "password"[3]. My father was justified in asking: "Umm, "passphrase" and "password"... Is that the same?"
   To computer-people the answer is obvious, to "normal" people it's not. It can easily confuse them. They may never try a command, because they think it's irrelevant for what they want to do.

   Once you've given something (a command, a dialog box, a certain view on the data, ...) a name, stick to it throughout the application. Especially with more than one developer, this isn't easy to achieve. The best way to succeed is in creating an **application lexicon** that contains all the terms the team agreed on. But test the terms on target users. They have to make sense to them, not to you. (See also 6.4.)

   It's easier to control the consistent use of the application lexicon if **all** text that will be displayed to the user is stored in a central file.

3. **Use terms that make sense to the user**
   People in every occupation evolve a language of their own. That's nothing bad. Problems arise when these internal terms leak out to someone outside the domain. In computer applications it happens frequently that users are confronted with tech speak. Try to use terms from the target domain of your application instead of computer terms.

Writing texts for applications is a qualification in its own right. Hire a Technical Writer if possible.

4.4 Errors

Human beings make mistakes. Always have, always will. Why should operating a UI be an exception?

Implications:

[3] I'm German, so it originally was "Passwort" and "Kennwort".

1. **Keep people from making mistakes**
   If you can prevent errors do it. A good example where this is successfully employed, are inactive, grayed-out menu entries for commands that aren't available at that moment.[4]

   Another situation in which you can avoid errors is when users are entering data, e.g. the contact data of a person: If possible let the user choose from a list of options, rather then letting him enter freeform text. In our example the date of birth springs into mind. Let him choose day, month, and year from three lists. Offering only a text field increases the chance that the user forgets a digit or violates the correct format.

   Don't ask for more information than you absolutely need. Provide sensible defaults, if possible.

2. **Error messages should be polite and helpful**
   Users don't commit errors on purpose. There's no need to bark at them or deliver tech barble that serves for debugging but not for an end user. For real "Error: 503", what is that supposed to mean? What about "Error: Division-by-0. Please check table cell E34."? That's more helpful.

3. **Always, at any costs, preserve the user's work**
   Provide **Undo / Redo**-functionality for everything. Make sure it works, because people will rely on it. If an action is not undoable, say so clearly beforehand and suggest a backup-copy.

   "User's work" can also be entered data in a dialog: The recently replaced vending machines for tickets for german railway were very unforgiving. Once I entered data to buy a ticket and then inserted my EC-card the wrong way[5]. The stupid machine returned my card and also returned to the start screen. Everything I specified was gone! You have to know, that the machines were cumbersome to use and incredibly slow - way over a second reaction time for every new screen. That makes such behaviour doubly inacceptable.

Taking common human factors into account is a step into the right direction. But as there are many differences from individual to individual, we're not yet done walking.

5 User-Centered Design (UCD)

User-centered design tries to optimize the user interface around how people can, want, or need to work, rather than forcing the users to change how they work to accommodate the system or function. From Wikipedia - [3]

This goal is achieved by involving users in the entire production process - from the first design drafts up to the ready-to-ship-application. It starts with a careful analysis of:

1. **Who are your users?**
   The most important distinction is experience: What backgrounds do they have? Are they experts in their domain? With computers?
   Other points: Age, gender, occupation, cultural background, possible disabilities, ...

[4]Please **never** remove a menu entry. That would be messing with the user's spatial memory. Graying out is a clearer feedback that the command exists, but not for the current situation.

[5]This only happened because there was no drawing next to the card-slot, telling the right way to insert. This drawing was on the screen some 20cm away from the slot. When standing in front of the slot, you can hardly see the screen. Here the developers made wrong assumptions about my focus.

2. **What is their task?**
   What do they want to achieve? What are the most important tasks? How often will they use your program? Will they receive training? Are other applications involved in the overall task?

5.1 Identifying your users

Sometimes you already know who the user will be, e.g. because you're hired by a client to write a specific application for her/her employees. In the latter case it is important to note that the client (to whom you sell) is not necessarily the user (who will work with your product). Try to get to know what the actual users are like.

In other cases you have to specify the target group yourself. Unfortunately for many projects the target group isn't really defined. Mainly because "everybody" has a lot of appeal. The sad news is, that when you target "everybody" you often actually target no one. If you target "everybody", everybody will be a little dissatisfied with your application and may switch to one that is more tailored to her needs. It's better to choose a target group and focus on making the application right for them.

5.2 Identifying their tasks

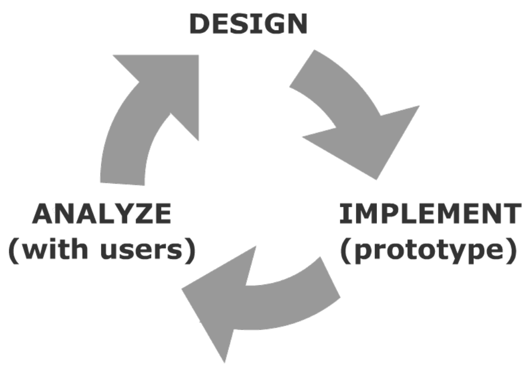
Once you spotted your future users, you can go about researching, what they will want to do with your software. What are functions they'll use heavily? What is less important? Ways to find out about that include the following:

- **Site Visits** - go and observe your target users in the enviroment they will use your product in
- **Questionnaires** - ask people in text form about their task, what applications they use, what they like / dislike about them, ...
- **Interviews** - similar to questionaires but more personal and you can ask users to explain when something is unclear

During all this, think in terms of "What will the user be able to do". Don't yet think about how you'll implement it.

When you're clear about your target user and his task, you enter an iterative process.

5.3 Iterative Development



From all your gathered information you design an application and implement a prototype. You test the prototype with real users to check what works and what does not.
You refine the design and implement a new prototype (or improve the last one.)

The importance of early testing cannot be overestimated. Usability - just like security - is not something you can “add” in the end. It’s not only about moving some buttons around, but about the whole concept of interaction.
It’s easier to let go of dear - but inappropriate - ideas, when you haven’t already put much effort in.

A prototype needn’t be software! For your first tests think of:

- **Paper Prototyes** - drawings of the screen in various stages of interaction
- **Post-It Prototypes** - can simulate menus, message windows, dialogues, ...
- **Photoshop/Gimp-MockUps** - more realistic but still without function

The more iterations you can break the development into, the better. Continue until you’ve developed the final application. With each iteration the application gets more concrete and the focus shifts more to details.

As you’ve probably noticed, user tests are the key element of user-centered design. That’s why I want to give you a little teaser of what a user test can look like.

6 How to run a User Test

Disclaimer: This is only a minimal version of a user test. A professional usability test is more sophisticated and definitely includes more users. But I found that even if you get feedback from only a single person, you know more than you knew before. So one is better than none. Two or three test users in each iteration will do a good job. Six to eight and you can be quite sure that you won’t miss anything important.

So this is the simplistic start into a future full of enchanting UIs:

6.1 You need:

- a test conductor[6]
- test users
- a test environment - for most cases: A quiet room
- equipment to record the test session
  (must: pen & paper; optional: video, audio, screen recording, ...)
- test material

Of all this, the test material is the trickiest. What to prepare depends on the test itself and will be explained in the according section (6.4).

6.2 How to choose a test conductor

If possible

- pick someone friendly, who is “good with people”
- pick someone who did not take part in developing the application
- or at least a developer who can control her reactions very well

[6]The test conductor runs the test.

Try to get people from your target group to participate in a test. (You know who your target group is, right?)

To be realistic: Unless you work in a company, most of your test users will be family members or friends. That's okay, as long as they are not afraid to tell you the truth about your application.

Ah, and prefer talkative people. You will learn more from them.

6.3 Treat your test users well!

That's the least you can do. Apart from the time they devote to you, they risk ridiculing themselves in front of you. It is very important in any kind of usability test to convey to the user, that **not he himself is being tested, but your application**.

Treating your test users well also includes:

- preparing well
- checking all equipment beforehand
- being punctual
- giving realistic estimates of the duration of the test while recruiting
- making sure all data is anonymized and not trackable to a certain test user
- a giveaway - needn't (but can) be money, a chocolate bar, a mug with the project's logo, ...

6.4 The test itself

In the beginning welcome the test user and give him some minutes to get aquainted with everything. You can begin when the user is relaxed. Explain what is going to happen and make sure the user understood what he's supposed to do.

There are a lot of ways to test users. We will pick two different test objects in order to look at two ways of testing.

1. **Test object: Application lexicon**
   Let's assume your team created a lexicon with the terms that will be used in the application-to-be. Then you could present each term individually on paper and ask the test user what he associates with it. When the term is a command, ask what the user expects to happen, when the command is executed. Bring an open mind. You might need to change some terms.

2. **Test object: Working software**
   For working software I recommend a technique called "Thinking Aloud". During a "Thinking Aloud"-test an observer, i.e. the test conductor, sits next to the tested user and takes notes of all occurances. The test user is asked to put his thoughts into words. Just like a sports reporter in radio commenting on a soccer game, he should comment everything he does: How he perceives things on the screen, what he guesses how things work, ...
   Using this technique often gives very surprising insights.

   After introducing the test tasks the conductor is as passive as possible. She shows neither approval nor disapproval. Test users often seek the conductor's help, but she shouldn't give any unless it's obvious that the user is completely stuck and that the test cannot continue without aid.

   If a developer conducts the test herself, she must be able to control her reactions very well. It can be quite painful to see that users don't grasp the concepts, one has so carefully devised.

But it would pretty much ruin the test if she jumped up and screamed “THERE!!one! In the bottom left corner! Why don’t you see this?”

Create concrete instances of common tasks as test material. Let’s look at a possible test task for an application to play mp3s:

(a) Play the song “Sad Robot” which is in the directory “home/music”.
(b) Create a playlist containing all songs in this directory.
(c) Delete the song “SchniSchnaSchnappi” from the playlist.
(d) Save the playlist as “MyPlaylist.m3u” in the directory “home/shared/”.

The test tasks should be tasks taken from real life. Also test the test’s setup (i.e. with a collegue) to make sure it’s possible to fulfil the tasks and to estimate the time needed.

For any kind of test you will get a more realistic feedback if you don’t tell the test user about all the effort you put into the application. If you do, test users probably won’t want to disappoint you. Especially if the test user is your mother.

7 Summary

Users are often forced by applications to figure out the application’s UI rather than being able to concentrate on their work. One of the reasons is that although we (as developers) like to think we know what users want, we actually have no idea unless we ask them.

Of course we need to know whom to ask and what questions. For that we need to specify our target users and get to know them and their tasks.

The earlier and the more often we ask, the better will our application be tailored to the users’ needs. When we get it right at the beginning, we safe ourselves a lot of trouble towards the end. On top of that we’ll have an application with a fairly decent UI and who knows, in time we may enter history as the successor of Apple :)

8 Do you want to know more?

8.1 Literature

- **“The Design of Everyday Things”** by Donald A. Norman
  A little older and sometimes repetetive but nonetheless lays the groundwork for usability awareness in general, albeit not explicitly about software.

- **“GUI Bloopers”** by Jeff Johnson
  Focused on software and websites. Contains many examples and rules that are applicable immediately.

- **“Don’t Make Me Think”** by Steve Krug
  Focused on websites but widely applicable. Great passages about user tests.

8.2 Links

- **User Tests**
  - General
    `http://www.infodesign.com.au/usabilityresources/evaluation/default.asp`

  - Real-Life-Example: Informal test of a CD-ROM
    http://www.webpagecontent.com/arc_archive/120/5/
  - Real-Life-Example: Test of an API 1
    http://blogs.msdn.com/stevencl/archive/2005/05/05/415016.aspx
  - Real-Life-Example: Test of an API 2 - Conclusions
    http://blogs.msdn.com/stevencl/archive/2005/07/11/437598.aspx

- **Random Sample of UI Guidelines**
  - Apple
    http://developer.apple.com/documentation/UserExperience/Conceptual/OSXHIGuidelines/index.html
  - KDE
    http://developer.kde.org/documentation/standards/kde/style/basics/index.html
  - Windows
    http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnwue/html/welcome.asp
- Collection of articles about UI design principles:
  http://developer.kde.org/documentation/design/ui/index.html
- Linux Usability Report (as an example of a usability review):
  http://www.relevantive.de/Linux-Usabilitystudy_e.html
- You're an OSS-developer? Get usability-feedback:
  http://www.openusability.org/
- You always do the opposite of what your told? Here are the golden rules of Bad Design :)
  http://www.sapdesignguild.org/community/design/golden_rules.asp

References

[1] Everything in 8.1. And additionally:

[2] Lectures: Designing Interactive Systems I + II, Current Topics of Human Computer Interaction
all given by Prof. Dr. Jan Borchers, Head of
The Media Computing Group at RWTH Aachen University
http://media.informatik.rwth-aachen.de/tiki-index.php/

[3] Wikipedia - User Centered Design
http://en.wikipedia.org/wiki/User_Centered_Design