Version 0.2

Enrico Zini (enrico@debian.org)

Zen and the art of Free Software: know your user, know yourself

Many free software projects don't seem to have a direction, and seem to proceed randomly in the dark. In order to better direct your project, you have to know your users, and you have to know the most important one among them: yourself.

The paper introduces some cheap, stupid and tremendously useful user-centered design techniques that can be used in the free software world, and some unexpected outcomes of the gained insight.

In Free Software there's a strange Heisenberg-like principle, in which you're both part of the developer and part of the user community. Better insight of your users, can turn out to be better insight of yourself.

You goals may be the same as your user's, and often they may be different. Why would we want to develop something that isn't addressing our goals? Why would we want to offer users something that isn't addressing theirs? Confusing goals bring to frustration; uderstanding them this brings to peace, harmony, and Total World Domination!

Introduction

Dear fellow free software developer,

first of all, thanks! Thanks a lot! Whoever you are, whatever the reasons you may have for developing free software, just by uploading source code to the 'net you're adding a gear to the big machine that gets my work done.

More than that, you're adding letters to a great alphabet that me, you and thousands of other people can use to create new systems, just as with the letters of the alphabet you can create new poems. No wonder Free Software has been declared World Treasure by UNESCO in November, 2003![see http://www.fwtunesco.org/welcome.html and http://www.fwtunesco.org/atmlist/freesoftware.html]

But what if I told you that you are shaping the society?

What if I told you that you're shaping your own identity?

This paper is written for free software developers, but users are warmly invited to stay and listen.

In the first part („Software and Society") I'll introduce some concepts which can be used to understand social aspects of software.

In the second part („Social Specifications") I'll show some practical techniques on how to plan and specify the social aspects of your software.

In the third part („Social Design") I'll show some useful concepts and tools you can use when designing the user side of your software.

In the last part („Social Testing and Debugging") I'll try to show some practical techniques on how to assess and possibly correct some of the social issues of your software.

I write this paper for four reasons

- For developers to pay attention at the social aspects of their software, possibly creating software which better fits the needs of their users.

- For users to pay attention at the social aspects of the software they use, and to start taking an active part in shaping it for their real needs. Free Software explicitly gives exactly this power, and it's a shame that we hardly harness it

- For bringing up happiness and harmony among developers and users.
  In Free Software, developers are often part of their user community, but don't always share the same goals of the other users. This causes frustration, misunderstanding, anger, and flamewars. Understanding this brings happiness, harmony, enlightenment, and Total World Domination.

- For me, to clear my mind about knowledge and ideas that I've been absorbing for some time, and I now want to try to put into some use.

Before I start, an important notice:
please consider this paper as version 0.1 in a „Release early, release often" scheme. Its contents are not intended to be a conclusion: they're intended to be the beginning of a long work.

Of course, everyone is welcome to join!

Related work

This paper puts together informations collected from various different places.

The part about situated action and situated identity comes from the extremely interesting and much wider work of prof. Giuseppe Mantovani[1].

Most of the part about what I called „social specifications" come from the excellent „Inmates“ book of Alan Cooper[2], which also tells a big deal about frustration, and contains the 12 magic rules of software politeness.

Beyond Cooper, there is a wide literature on user and task analysis, with a lot of notations, methods, processes and tools; a good presentation of it is in the book by Hackos and Redish[3]. The ConcurTaskTrees resources [4] are also a good starting point from where to explore notation and tools.

Nielsen's Euristic Evaluation comes of course from the Usability work of Jakob Nielsen[5], whose Useit website[6] is a very good usability resource. If you're into unsability, don't miss the POET[7] book by Donald Norman, which is a very good place to start learning things and a very nice read.

The Flanagan Critical Incident Technique comes of course from the work of Flanagan[8] in the field of ergonomics.

The 7 ∓ 2 magic number is explained in Miller's works[9] in cognitive psycology.

Software and Society

In this chapter I'll introduce a few concepts which have a key role in understanding social aspects of software.

To avoid making this paper hard to read, I'll try to keep the number of new concepts small; however, please pay attention to them: words, and the concepts encloses, are in our brain powerful cognitive tools with which we understand and control reality.

In Orwell's 1984, the regime of Oceania reducing freedom by reducing the amount of words to be used. Almost 2000 years before Orwell, someone wrote:

> *In the beginning was the Word,*
> *and the Wordwas with God,*
> *and the Word was God.*

(john1:1)

Could it be explained any clearer?

Situated action

Take a talk listener at a conference. He's sitting, watching the speaker, listening carefully, sometimes nodding. How much can we tell about him? You probably think he's interested in the talk. But what if I tell you that the speaker is going to receive the listener for a job interview in a few days? What if I tell you that the speaker is a gorgeous woman and the listener is single?

In order to interpretate the meaning of an action, we need a context. *Situated action* is the term that identifies an action considered in its specific context.

This is no breaking news, really, but for some reason we tend to think in absolute terms. The sky is blue. Vim is hard to learn. Users are clueless. This needs to be broken in order to work beyond the technical details of our software.

Just like we need to have runtime informations to understand bug reports on tecnical issues, it's important that we look at the situated actions when we try to understand non-technical issues like frustrations, difficulties in picking up concepts, lack of documentations, „suck“ reports („your software sux!“).

Looking at situated action is like looking at users and usage of your software including runtime informations. It sounds tremendously useful, isn't it?

Situated identity

We can go further and talk about *situated identity*: are you the same person when you go out with your friends and when you go out with your parents?

Our identity expresses itself in different ways as we continuously adjust our perception of ourselves with the frame of what we are allowed to do and say and be in a given context. This is natural, it happens all the time, and makes us wonderfully adaptive and successful in a wide variety of conditions.

Consider now that the whole point of software is changing our notion of what we can do: I install a piece of software to be able to do something that I wasn't able to do before, or that I could do differently. If software changes our notion of what we can do, then *software itself situates the user*!

Have you noticed that when run vim you are probably a bit different than when you run OpenOffice? Different softwares give you a different perception of what you can do and what you can't do, and this affects your situated identity. Software can make you feel smart, agile and powerful or goofy and vulnerable. The effects will be non-permanent until the advent of computerized body implants :)

If it can happen with editors, think of what can happen with communication software: we already reckon that people write „your software sux“ in a mailing list, but they will rarely attack you frontally like that in real life. In a mailing list you don't have empathy, while on real life you hopefully do. In a mailing list you don't see who's around, in real life you usually do. the situation changes, your identity changes as well.

Could it be that Outlook users are more likely than mutt users to be seen as fancily dressed jerks that get into conversations at the wrong time? Could it be because mutt has good threading support and no HTML mail support while Outlook has poor threading support and a very powerful HTML mail composer?

How much can we say it's only the user's fault if s/he appears as a jerk? Is s/he a jerk in real life, too? Won't s/he be angry if s/he'd discover that s/he's got a jerk reputation mainly because of the mail software that came installed on her/his PC? Wouldn't s/he be frustrated if s/he hadn't the possibility of switching to some other software?

Being aware that software contributes to the situated identity of its users gives us developers a lot of power, and a lot of responsibility. But there's more: we could start designing software that supports an identity *that we decide*. We can even take the challenge to design software that supports our own identity!

A nice thing of doing this, is that the process overlaps with the one of discovering who we really are :)

Frustration

n. The feeling that accompanies an experience of being thwarted in attaining your goals

from WordNet 2.0 dict database

We use software to attain some goals. To put it in a simple way, if we can't attain it, we get frustration from using that software.

Like segfaults are a good indicator of a technical bug, frustration is a good indicator of a social bug: the software fails when you are unable to attain the goals you were expecting to attain.

This is very important to know! As a user, in this regard, you have a possibility of being aware of the cause of your frustration before you convince yourself you are wortless with computing and start reading mail only if someone prints it on paper. As developers, we have to be able to spot and investigate a social bug, and to justify the patience and care needed to interact with someone who is frustrated because of us.

Coping with frustration should be one of the first things people get told when put in front of a computer for the first time! As a user, what I do now when I feel frustrated with software is:

- ask myself what goal do I have. This is not easy at the beginning, but after a while you get good at it
- see what tasks I was performing in order to attain my goals.
- if I think I chose the tasks correctly (my strategy was correct), then I isolate the reason why I'm stuck and file a „wishlist“ bug report in which I explain what I wanted to do, how I wanted to do it, where I got stuck and a possible way around the problem.

In the end, I still don't get to my goals, but I do feel better and I have also made a valuable contribution... if I've been wise enough not to leak some bad words from the frustrating moments into the bug report :)

Social specifications

We techies all more or less got to love reading feature lists. We drool on reading them, tasting the feel of power and feeling our range of possibilities getting larger and larger item after item. We got so used to feature lists that we even design software around feature list.

Let's see a feature list-like specification:

- 2-wheeled vehicle
- small
- agile
- carrying up to 2 persons

Doesn't it seem to lack something? What if before the feature list I added this:

- motorbike for single parents to go to work in the traffic

Wouldn't it feel like it would lead to a much, much smarter design?

The feature list lacks two important design parts: users and goals. It risks building something cool, but that would serve no one.

Users

Creating a *persona*

First thing: ask yourself: *„who are you developing this for?“*.

When you start designing a piece of software, you are trying to solve someone's problem. It is imperative that you know very well what kind of person you're working for, and keep it clear during all the design and implementation. Create a prototype target user (Cooper calls it „*persona*“), give it a name of a person that doesn't exist, write down its profile on a piece of paper and stick it to the wall. Write it in the beginning of your README file. Put as many details as you can, even pointless, but anything that can make you remember who's your target.

It can be that you are designing for yourself; but who will you be using that software? Which of your situated identities are you addressing? It could even be a new identity you want to give to yourself. Even so, nothing changes: create a persona, give it a name, write down its profile in great detail, add some pointless details even, and stick to it.

Just one small note: *a persona must not be any real person*. You are usually address a type of user more than a single user, so your persona should reflect that stereotype. Real persons have unique quirks, and if you take them into account you risk losing time and efforts in something pointless. Choose your stereotypes wisely. Whatever kind of user you are designing for, your persona must be the *average sample* of that kind.

Using a persona

The persona you created has an extremely valuable potential: it describes what kind of person is supposed to be happy and satisfied when using the software you're designing. It will be the main scaffold of your design: all other design and implementation choices will depend on the persona.

Pay extra care in not changing the persona during further development stages: it's quite instinctive to say „Sure, Mr. Aldo Waldorf is a 60 years old man who just wants to store his digital pictures, but maybe he's curious to see a raw dump of the EXIF details: let's implement an Advanced menu with stuff like that“. It's really not „Aldo Waldorf“, your prototype persona, that needs that feature: it's you! Aldo will most probably see „Advanced“, try the option, see some incomprehensible gadget and conclude he's not advanced enough for computers. That feature is more likely to make Aldo feel stupid. If you tend to bend and stretch your persona, you are creating in Cooper words an *elastic user*, which justifies any choice you make and renders the entire point of having persona completely useless.

Please read again the discussion about Aldo in the previous paragraph. See how it's easy to reason about users when you have a persona defining what concept of user you are doing?

Using personas rocks! The concept was proposed by Cooper in 1999 already. Why does it take so long for us to start mastering the idea?

Goals

After defining the persona, the next step is to define its *goals*. Goals are what will define the functionality of your software. Goals tell what problems your software is supposed to solve. Goals are your challenge as a developer. Goals are your reason to write software in the first place.

Cooper defines 4 interesting kinds of goals:

- *personal goals*: the main goals of the person using the program. They may even be unrelated to the program domain, such as „don't feel stupid“ and „don't make errors“, but they are the main reason your users use your program. They have priority over everything: if you miss personal goals, your program just won't be used.
  When you get to understand goals a bit better, try to remember how many otherwise very good pieces of software miserably stay unused and disappear. One of the most likely reasons for it, is that they were missing their users' personal goals.
- *work goals*: the goal for which the person is requested to use the program. They may be something like „increase sales“ or „submit a paper to the conference“. Work goals are said to be *hygienic* goals: they are required to use the program, but they are not the users' motivation of using the program. A program that targets work goals but misses personal goals, fails.
- *practical goals*: practical goals are required goals needed to reach personal or work goals. If your personal goal is to view a nicely rendered 3D model of a scene, before getting that you'll have the practical goal of creating the

scene model. Practical goals are hygienic, too: they are required, but they aren't the main cause for motivation.

- *false goals*: there are thing we tend to take for granted and strongly influence our design, but which are not really goals. For example, „use few CPU cycles“ can be a false goal which makes our application more dumb. Word processors and smart editors got away with this false goal and do use CPU cycles while we type, providing features like error highlighting, word completion, method name completion that we all tend to love.
  Even „be easy to use“ can be a false goal: if you're designing a plane cockpit, you don't want it to be easy to use, but you want it to be efficient: you don't mind the pilot requiring 4 years of training to use it, as soon as she doesn't make errors and can spot and correct anomalies in the most efficient way.

Try to work out a list of the various goals of your persona, and try to sort them out in the categories of „personal“, „work“, „practical“ or „false“. It may not be easy at first, because we're used at a huge level of confusion in this field; but once you built yourself a good picture of the goals you are addressing, you'll discover that the rest of the design, and even the implementation, will be just a downhill ride.

Social design

Tasks

For most small software projects, once you get persona and goals right you already have a clear idea of all the rest of the design and implementation, and will code your successful application with great enthousiasm and satisfaction.

If you are building a bigger application, however, and especially if you are building an interface for something new, for concepts for which no interaction has been implemented and tested yet, you may want to take an additional step: *task analysis*.

Performing task analysis means working to figure out, for every goal, what tasks your persona will need to perform in order to attain it. Here are some examples of task analysis you could do:

- investigating how people work towards those goals nowadays, maybe without using a computer.
- decompose tasks in subtasks (hierarchical task analysis), both to understand them better and as a basis to lay out the various parts of your interface.

There are notations and tools to assist in task analysis and other related steps [4].

If you want to know more, you can find a lot, and I mean a lot of techniques, suggestions, examples about user and task analysis on [3].

Software politeness

Alan Cooper provided a very nice checklist of for designing good applications, tailored like politeness rules for clerks and waiters: the 14 principles of *polite software*:

1. **Polite Software Is Interested in Me**
   trying to remember my preferences and to adapt to my habits; the proposed defaults, for example, could be computed based on the values I input more frequently
2. **Polite Software Is Deferential to Me**
   error messages should not insult me, but soggest the right way of doing things. Likewise, tooltips, labels, hints and other messages should appear as suggestions on how to do things in the right waay, not as orders to make lusers comply
3. **Polite Software Is Forthcoming**
   software should be ready to answer to my requests doing all what is necessary: if I ask for the soup at the restaurant, the waiter will also bring a spoon even if I didn't explicitly ask for it
4. **Polite Software Has Common Sense**
   software should satisfy users' goals in the most appropriate way possible, without digressing in useless things
5. **Polite Software Anticipates My Needs**
   where possible, it'd be nice to anticipate future options of users and get somehow ready for them. For example, a web browser could preload the text content of every link target in a page, so that whenever we click on one, we can see the link contents right away.
6. **Polite Software Is Responsive**
   if I have an obvious need that can easily be understood by the computer, the computer could provide it even if I don't ask for it explicitly
7. **Polite Software Is Taciturn About Its Personal Problems**
   many programs continuously interrupt us with popups, messages and useless informations about their internal state, which often have no purpose besides disturbing or annoying the user
8. **Polite Software Is Well-Informed**
   it shouldn't offer to do impossible things: web links bringing to a 404 error could be rendered in a different way, so that we can just avoid to click them
9. **Polite Software Is Perceptive**
   Software should be able to perceive my needs from my usage patterns, and behave accordingly. if every time I open a window I maximize it, after some time that window could open already maximized
10. **Polite Software Is Self-Confident**
    there is no sense if a program always asks „are you sure?“ for any order it gets: it would be better if it just did as said, and then allowed us to undo possible undesired actions.
11. **Polite Software Stays Focused**
    a program should ask a thousand questions for every operation, but only ask

the minimum it needs and allow to care for the details only when necessary

12. **Polite Software Is Fudgable**
in many processes, the possibility of making exceptions to rules can be esential to be able to cope with the mundane complexity. For example, if I am asked 100 pieces of informations but I can only enter 99, the computer should not totally abort the transaction, but could remember the 99 items I entered and allow me to insert the 100th at a second time. Likewise, sometimes it could be vitally important that an urgent print job could jump in front of the print queue. Obviously, software should also allow to verify that exceptions keep being exceptions and don't become the rule

13. **Polite Software Gives Instant Gratification**
software should start to do what we ask as soon as possible, as long as it can, without waiting for us to enter all data

14. **Polite Software Is Trustworthy**
if we use a software, it's to make it part of our job: it shuold therefore be possible to trust the result, without needing to check it every time

Summarized from A.Cooper, „The Inmates are Running the Asylum“, SAMS 1999

Social testing and debugging

Once your software is implemented (or even while you're implementing it, or trying out prototypes), there are some nice ways of assessing its behaviour. I'll present here a couple of simple evaluation heuristics, a nice investigation technique and a magical number that noone should ever forget.

Nielsen's euristic evaluation technique

Famous usability expert Jackob Nielsen (you probably know its usability website www.userit.com) gathered over time some usability wisdom in 10 simply euristic rules that form a useful checklist with which a program can be evaluated:

1. **Visibility of system status**
The system should always keep users informed about what is going on, through appropriate feedback within reasonable time.

2. **Match between system and the real world**
The system should speak the users' language[1], with words, phrases and concepts familiar to the user, rather than system-oriented terms. Follow real-world conventions, making informations appear in a natural and logical order.

1 An example of this problem is in: http://www.prdomain.com/companies/a/amd/news_releases/200307july/pr_amd_nr_20030707.htm

3. **User control and freedom**
   Users often choose system functions by mistake and will need a clearly marked "emergency exit" to leave the unwanted state without having to go through an extended dialogue. Support undo and redo.

4. **Consistency and standards**
   Users should not have to worry wonder whether different words, situations, or actions mean the same thing. Follow platform conventions.

5. **Error prevention**
   Even better than good error messages is a careful design which prevents a problem from occurring in the first place.

6. **Recognition rather than recall**
   Make objects, actions, and options visible. The user should not have to remember informations from one part of the dialogue to another. Instructions for use of the system should be visible or easily retrievable whenever appropriate.

7. **Flexibility and efficiency of use**
   Accelerators -- unseen by the novice user -- may often speed up the interaction for the expert user such that the system can cater to both inexperienced and experienced users. Allow users to tailor frequent actions.

8. **Aesthetic and minimalist design**
   Dialogues should not contain informations which is irrelevant or rarely needed. Every extra unit of informations in a dialogue competes with the relevant units of informations and diminishes their relative visibility.

9. **Help users recognize, diagnose, and recover from errors**
   Error messages should be expressed in plain language (no codes), precisely indicate the problem, and constructively suggest a solution.

10. **Help and documentation**
    Even though it is better if the system can be used without documentation, it may be necessary to provide help and documentation. Any such informations should be easy to search, focused on the user's task, list concrete steps to be carried out, and not be too large.

Jakob Nielsen **quoted from**
http://www.useit.com/papers/heuristic/heuristic_list.html

Flanagan Critical Incident Technique

John Flanagan developed the critical incident technique (CIT) to ***identify behaviors*** *that contribute to the success or failure of individuals or organizations in specific situations.*

To analyze a situation using CIT, a researcher first asks people familiar with the situation for a recent example of effective or *ineffective behavior — that is, a **critical incident**. A critical incident is determined from answers to the following questions:*

- *Describe what led up to the situation.*
- *Exactly what did the person do or not do that was especially effective or ineffective?*
- *What was the outcome or result of this action?*
- *Why was this action effective, or what more effective action might have been expected?*

The researcher uses the answers to these questions to identify themes represented by the incidents, and asks other involved parties to sort the incidents into proposed content dimensions. These retranslation exercises help identify the incidents that represent different dimensions of the behavior under consideration.

*The critical incident technique has been applied in studies of numerous occupations, including aircraft pilots, dentists, teachers, plumbers, and infantrymen. The **Society of Industrial and Organizational Psychology** recently recognized Flanagan's 1954 CIT article as the most frequently cited article in the field of industrial and organizational psychology.*

from http://www.air.org/overview/cit.htm

We've been having a debugging technique for social aspects of software since 1954! Let's use it! (just remember to ask questions a bit more nicely than the samples above :)

The magic number 7∓2

7∓2 is a magic number related to our brain which is known since at least 1956 [9], and roughly represent the average capacity, in *items*, of our short-term memory.

Spelled in more familiar terms (uh, well, for software developers :) our brain processes *items* (arbitrarily complex but atomic pieces of informations) using a cache (the *short term memory*) which size is something like 7∓2 items.

Never forget this number because it has tremendous connection with the cognitive load of the informations you present to the user. For example, tasks which require to process more than 7∓2 items at a time are perceived to be difficult.

Try an experiment: write a perl script to print lists of totally unrelated items (for example random small digits or random dictionary words), and read the output at increasing lengths. If the size of the list is under 7∓2 items, then you'll be able to easily remember all the list after reading it just once. As soon as the list gets longer, items will start being pushed out of your brain cache and remembering everything will be a more difficult task.

Of course our brain can process more than 7∓2 pieces of informations. We do that with a trick called *recoding*, which consists in creating a new item which groups related items together, and then we process the macro items. The long term memory can be thought as the collection of the recodings we have created over our lifetime.

Given this, we can learn many lessons:

- Don't expect users to be able to remember more than 7∓2 pieces of informations at the same time. Cognitive load works like phisical load: if you load yourself too much, you get tired quickly or simply can't carry the weight.
- Don't present more than 7∓2 similar pieces of interface together. If you need to, try to group them with some meaningful relationship.
- Give items a chance to form recondings: present related informations visually together, so that it can be thought as a single item.

This is only the beginning. 7∓2 is an omnipresent magic number in cognitive tasks: try to to see where you can spot it, and you'll get to be able to estimate the mental difficulty of cognitive tasks just as easily as we are able to estimate the physical loads of physical tasks.

Bibliography

[1] Giuseppe Mantovani, "New Communication Environments: From Everyday to Virtual", Taylor & Francis, February 1996

[2] Alan Cooper, „The Inmates Are Running the Asylum: Why High Tech Products Drive Us Crazy and How to Restore the Sanity“, Sams Publishing, 1999

[3] Hackos, Redish, „User and Task Analysis for Interface Design“, Wiley&Sons, 1998.

[4] Fabio Paternò et al., „The ConcurTaskTrees Environment“ website, http://giove.cnuce.cnr.it/ctte.html

[5] Jakob Nielsen, „Ten Usability Euristics“, http://www.useit.com/papers/heuristic/heuristic_list.html

[6] www.useit.com

[7] Donald Norman, „The Design of Everyday Things“, Basic Books

[8] http://www.air.org/overview/cit.htm. also see Fitzpatrick, R. and Bolton, M. (1994) „Qualitative methods for assessing health care“ in Quality in Health Care 3: 107-113

[9] George A. Miller, „The magical number seven, plus or minus two: Some limits on our capacity for processing information“, 1956, Psycological Review n.63 pp. 81-91, http://www.well.com/user/smalin/miller.html