

Complexity

“*Easy to use is easy to say, but hard to do*”

Jim Allchin – 9/4/97

While I think everyone acknowledges that our software and PCs in general are hard to use, I don’t think the magnitude of the problem is understood at all. No other consumer product in history has been so hard to use and so fragile. Yet computers and software have continued selling despite this. It is an amazing testament to all the benefits that personal computers bring that consumers are willing to put up with the vast array of associated problems.

All of us have our horror stories about spouses, neighbors, family members, *ourselves* where we wasted a weekend (or longer!) trying to get something to work. We are supposed to be experts, but even we can’t get things to work properly. (Any one who gets ISDN working without getting help from someone else deserves an award.) It is a total embarrassment. Craig Mundie said to me recently that he never touches his machines in his home except at the beginning of a weekend because it knows it might take the entire weekend to recover from whatever goes wrong.

What is fascinating is that when you talk to average users, they assume that *they* aren’t smart enough or that *they* are doing something wrong when the computer doesn’t work properly. However, if those same people were dealing with other consumer appliances, they would more than likely say that the appliance was designed wrong. There is even a culture growing dealing with “idiots” and “dummies” that have to work with computers. This is an amazing social phenomenon. Eventually consumers will wake up with the equivalent of Ralph Nader taking the lead.

So, is it just that software has gotten so complicated that we can’t make the system resilient and easy to use any more? Is it that the demands for features mean that we can’t focus on naïve users? Is performance or memory size more important than consistent operation? What are we going to do about the hardware companies that continue building unreliable PC products?

Our software is approaching massive size (and internal complexity). NT Workstation 5.0 will have around 27 million lines of code at Beta 1 – and more will be added before it finally ships some time next year. For perspective, the first version of NT only had around 6 million lines of code. In my view however, the size of the system has nothing to do with whether the system can be made resilient or easy to use. In fact, more software is required to make systems consistent and more reliable; it just has to be the *right* software.

Our products are becoming more and more feature rich. We have ways to “tweak” virtually everything. The amount of information known publicly about flags/fields stored in the registry and how to tailor aspects of the software is amazing. We present user interface dialog box tabs covering technical features that only a miniscule fraction of all users will ever need (or understand). You could argue that this is great --- it’s a Turing machine, you know – you can customize it to do anything. Unfortunately, we seem to look at these things as a computer geek does and forget about “the rest of us”. On the other hand, we have done some good work in areas where we have added features (e.g., spelling correction in Word) which do not add additional complexity. I believe you can simplify the experience for naïve users and still maintain the flexibility for power users.

What about performance? Is it ok to trade off consistency or reliability for performance? How about trading off consistency or reliability for smaller memory size? Is it ok to make the system super fast *most* of the time, if it doesn’t operate correctly (or crashes) in some rare cases? Microsoft has a disease dealing with these questions. I have been in many meetings where tradeoffs like this are actively discussed from billg on down. “We could win the benchmark.” “It saves 80K of memory.” This is a travesty. It would be better to not have a feature than have one that doesn’t work consistently 100% of the time.

So the question is: Is it possible to have a set of rules that guide the development of software that is more resilient, consistent, and easier to use? Unfortunately, software is still art and I do not know of any perfect rules, but I do believe there is a philosophy that can be adopted to start addressing our complexity morass. The rest of this paper covers this philosophy.

The thoughts below may seem very obvious. You may disagree with some of them. Nevertheless, I claim that following these rules would result in software order of magnitude less complex and friendlier to humans. The question is why don’t we follow them.

# Quality

From rec.humor.funny:

|  |  |
| --- | --- |
| **Multitasking**  You can crash several programs all at once. No waiting!  **Built-in Networking**  You can crash several PC’s all at once. No need to buy Novell Personal Netware or LANtastic to crash.  **Microsoft Network**  Connect with other Windows 95 users and talk about your crash experiences. | **PnP**  Plug and Pray (that it works)  **Multimedia**  Experience the immense sign and sound of crashing.  **Compatible with existing software**  It will also crash your existing software |

From KISS Software, Rescue Me! Product advertisement: “There are over 9,345 ways to trash your system.” “Windows in your TV? Who wants to boot your TV every few hours?” I could go on and on. Our products have a poor quality reputation.

Quality is about ensuring the system works consistently: every time, all the time. A key metric I think about is crashing. Our software should never crash – period. We ship software today that we *know* has bugs in it that can crash the system or that has architectural holes that would allow an errant application to crash the system. We are under “market pressure” and somehow are able to convince ourselves that the likelihood of the condition (usually very complicated) occurring is very small and so it is ok to ship the product and fix it in the next release.

The first building block for making a system easy to use is quality. Quality doesn’t make a system easy to use; on the other hand, you can’t have a good system without it. You can have a very hard to use system that has high quality. MVS or VM/CP represent such systems. Once you get one of these systems running they are amazing resilient to hardware and software problems. I spent time with IBM engineers adding code for some of the error recovery paths for VM/CP. Tons of code exists to handle faults in these types of systems. It’s not that these systems were designed correctly. If you changed a line of code you were required to add your initials to the comment field on that line. I would look at pages and pages of BAL assembler code where the comment field of *every* line had been updated. And not by a single person. In more cases than not, there would be 5 or more initials on the same BAL line. Code (and quality?) by brute force.

So everyone says they support quality (Mom and apple pie, right?). What should we change?

* We should tradeoff compatibility for quality if necessary. At a minimum if we know that being compatible with some feature (from the past) could cause the system to crash or get in some inconsistent state, then we should have an option that a user can select that prevents the compatibility, in favor of stability. Users then have a choice in how the system operates.
* No algorithms should be used which we know a priori could cause the system to fail or enter an inconsistent state. No benchmark, size, or schedule crunch should be allowed to come between quality and us.
* We must ensure that PC hardware doesn’t impede our quality efforts. The classic example here is ISA slots allowing non-PnP cards to be supported. But, it goes beyond this. We need to drive the industry to address areas that are likely causes of consumer issues and faults in the system. An example here would be the fact that PC-Card (or PCMCIA) devices can be removed without warning from the system. That’s like being able to rip a CD out of your audio system while it is still moving and playing. If we want to improve the experience for end-users, we must focus on improving the fragile hardware we have today in PCs.

# Resiliency

Ok, suppose the system doesn’t crash. But, does it recover gracefully from problems? Resiliency is about having the system recover automatically from environmental issues (e.g., server going away, DHCP server not found, etc.) or common user mistakes.

This takes a very specific designer mindset. You have to assume that if something could go wrong, it will and then program defensively for these problems. Too often I find designers programming for the case when everything works perfectly. When an error occurs, they simply give up, spew an error message to the screen, and return an error message to their caller (who of course does nothing with the error except return it to their caller, etc.).

Networking is perhaps the area where this is most visible. Our networking code is extremely fragile today. Virtually any problem (lost packet, server not responding after we have a connection to it, configuration error, etc.) either makes the system totally unresponsive (with no indication to the end-user what is going on) or worse yet it actually stops working correctly. This lack of resiliency is a problem for the operating system (e.g., networking errors during authentication, etc.), but it’s also a problem for applications such as mail, calendar, chat, browsing, etc. No other peripheral is more error prone today than the network. It takes special designs to deal with this.

How many times does your computer just sit there doing nothing after you entered a command. There is no disk activity; no screen activity; nothing. But, you can’t do anything since the application or the operating system is “locked up”. This is because the software isn’t handling the network (flakiness) correctly. The network will continue to be very error prone (e.g., different delays, outages, errors, node failures, etc.) for quite some time in the future. Moreover, the fact that we want to be mobile moving between wireless cells or connecting between very different speed networks means that handling networking resiliency must be a top priority for us. Architecturally, we need to learn about network errors in a standard way and then avoid multiple timeouts, nested retries, etc.

We do have parts of the operating system that are more resilient today than other parts. One example is scandisk that automatically “repairs” the system if there are disk inconsistencies detected. The issue today is that scandisk may not correct the problem totally. In fact, it could delete a critical file. Still it is better than not doing anything. Assuming that we must have a utility like scandisk (because of the file system design) we need to ensure that upper levels of the software can recover from perhaps poor deletion choices that scandisk might make. (Furthermore, as noted later, if we have to have scandisk utilities, they should run silently without introducing the user to any bizarre terminology or operational choices.)

Resiliency is about keeping the system running at all costs. An example of this is that if the NT 4 shell (or IE 4 shell on NT 5) dies, the desktop disappears for a second. But, the operating system restarts the shell quietly in the background. The system is resilient to shell faults. This is not the case with win95 or win98.

Ways to improve resiliency:

* Auto-configure. Do not let the user configure things. Networking is one of the most complicated areas of our system. Asking a novice (or even an “expert”) to configure a protocol stack is like asking a layperson to do surgery – it takes training and even then they sometimes make a mistake. If you can't auto-configure, then the system should probably be redesigned so that it can be auto-configured.
* Assume a machine is transiently connected to the network. Never allow the user interface to be stopped by a network outage. Switch gracefully (but visibly) from network mode to offline mode.
* Once an error is detected (e.g., network timeout), it should be sensed only once. That is, at most one timeout should happen until there is good communications established again.
* Design for error cases even if it means the regular operation of the system is impacted slightly. A good example here is out of disk space. Assume you run out of disk space at the worse possible time. Will the system still function correctly? Can the user still remove some space? If it takes allocating some free disk space as a buffer during normal operation (to ensure that the system is still totally functional) then do it.

# Terminology

For some strange reason we insist on trying to teach people a foreign language: *geektalk*. My Mom doesn’t want to learn another language right now. She just wants to track her church work, write letters, exchange email, track her budget, look for things on the Internet, etc.

Geektalk is actually one of our worse sins in interacting with a user. Our systems tend to produce more internal information than a user needs to know and in doing so we use terms that only a computer literate person (and in some cases programmer) would understand.

Turn on (note a naïve user thinks a “boot” is a shoe) any PC and just watch the words/characters fly by.

**24576 KB OK**

**SystemSoft BIOS for Opt Viper 557/508N 1.00 (2450-35) (Version 01.11)**

**SystemSoft Plug-n-Play BIOS**

**<F10> to enter Computer Setup**

Oh, I see…

Now of all of the above, what do you think a first time user *has* to know? What terms would they know? Why do we insist on telling a user all these things? I believe it is because the system is basically in debug mode all the time. We never know when a user might have to know the internals of the system to deal with a problem.

When our software starts running we are even worse. I can’t even begin to represent the insane number of terms we use. We use terms such as “FAT32”, “clusters”, “defragging”, “DHCP”, “DNS”, “WBEM”, “WAN Mini-port”, “ODBC”, “enable IP forwarding”, “bindings”, “NDIS Proxy TAPI Service Provider”, “Video Compression Codecs”, “Device Tree”, “devmgmt”, IRQ, DMA, “Launch Internet Explorer Browser” (launch?), “Insert ActiveX Monitor Control”, etc.

And our error messages are nothing short of amazing. Consider these error messages from NT 5:

* No IPSEC Policy DN.
* {A79C9582-19D1-11D1-91C7-C4252BDEA3A4} : Could not find an adapter.
* The Net Logon service terminated with service-specific error 3095.
* No Windows NT Domain Controller is available for domain NTWKSTA. (This event is expected and can be ignored when booting with the 'No Net' Hardware Profile.) The following error occurred: The RPC server is unavailable.
* Illegal instruction
* And on and on.

When a TV is sold the manufacturer doesn’t have to explain how to read the color bands (detailing resistance) on some resistor that is used. Why do we insist on trying to make computer people out of everyone? We are still making software for the hobbyist. We need to make software for the consumer. Do we want the user to debug the system? Is our software so unreliable that we have to be in pseudo debugging mode all the time?

Of all the issues raised in this paper, reducing information dumping and simplifying/removing terminology is the easiest to change. Every product should have a glossary of terms that is used throughout the product. We can introduce new (computer) concepts to end-users, but they should be done with great care. Developers and user education (e.g., help, documentation) should only communicate using layperson’s language and the approved glossary.

One other key point is that some areas of a product might require more advanced knowledge and therefore a richer set of terms. In this case there should be an advanced button (or tool) that permits access to these terms (otherwise they are hidden). In general though these advanced dialogs should be avoided. Why? Because having the concepts exposed at all makes it hard to avoid having some error/information message pop out at a naïve user (e.g., “can’t renew IP address because DHCP server is not responding”). It would be far better in my view to have a command line utility that adjusts certain parameters (as in debugging mode), then have this information exposed in a general UI. Then we would never have to even explain what something like a DHCP server was to an end-user.

# Consistency

Lack of consistency is a key problem in our user interface and our underlying conceptual model. Humans are amazingly adept at dealing with ambiguity and inconsistency – but only up to a point. Much of learning is about understanding rules. If A, then B. Anyone who programs knows that deep nesting of **if**/**else** statements is very difficult to debug. In every day life it is no different. If you have to memorize one exception after exception, it is very difficult to not make mistakes unless you are living and breathing the environment constantly. Teaching someone about the system deals with teaching him or her about the rules. You want the rules to be very simple: if you do this, then that happens.

Our systems have tons of footnotes on how it works. Let’s look at some examples.

**Inconsistent drag-drop**. When you drag a file from one place to another, it would seem reasonable that the file will be moved, not copied. That rule is not correct however. If you drag a file on the same disk, then the file is moved. However, if you drag between disks, the files are copied – not moved. If you drag an EXE, then a shortcut is created, and the file is neither copied nor moved – that is, unless the destination is a removable drive.

**Tray**. This area on the screen doesn’t seem to follow *any* rules. It holds mostly icons except for the clock that is. Some icons get double-clicks, some get single-clicks, some get right-clicked, and some don’t get clicked at all. Some icons have tool-tips, others don’t (it depends on whether the application window is selected or not). With IE 4, there is yet another area on the tray that operates totally differently than the other tray components. It is magic.

**Shortcuts**. Shortcuts are very inconsistent since they are not supported through win32 APIs. Shortcuts are kludges done on top of the file system. The result? Terrible inconsistency. Whether a shortcut is linked to a file or a folder, it is always treated as a file. Specifically, you can’t specify a shortcut to a folder in a path. So, shortcuts are “sort of” transparent, but not always.

We should not be rigid about consistency. There is a higher level principle of familiarity (with real –world objects/tasks) that I think is even more important than consistency across all aspects of the system. Nevertheless, we are not consistent enough today in our systems. We use different terms for the same task, object, etc. in different parts of our system (e.g., Emergency Recovery Disk). We need fewer concepts defined and few ways to do things. Concepts that we introduce should be consistent unless there is a overwhelming reason not to be.

# Familiarity

A key to allowing people to learn something quickly is to build on prior knowledge. We need to maximize the concepts and techniques in our products that users are familiar with from their real world experiences. The first time someone uses an automated washbowl, they look curiously around the sink for handles. Once someone explains how it works or they discover how it works independently, then from then on they will intuitively try putting their hands under the nozzle if they don’t see handles. Of course, in some countries the water flow is controlled through foot petals. And someone who is used to automatic washbowls will be surprised when the water doesn’t come on. (I know because this happened to me.) The point is that once someone learns something they become familiar with it and expect it to be the same in a variety of situations.

Our user interface and our conceptual model should attempt to model within reason tasks or objects that people already know. If there is an analogy that exists outside the computer realm that a culture understands we should use it in our systems. A good example of this is highlighting in Word. This is something that virtually all people are familiar with. Using yellow as the default and calling the feature highlighting is easy to understand for most everyone.

Designers should not force fit metaphors if the computer operation isn’t a close match to the real world. Strong metaphors (e.g., phone) need to behave as much like a phone as possible – because users really know how a phone works and they will map their knowledge appropriately onto the computer interaction.

When I am writing a letter, I am writing a letter. I usually don’t want to deal with anything else. A new mail indication might interrupt me and I might switch to reading mail, but that is no different than being interrupted by a phone call. In both cases, though I am doing a single task. This argues for more full screen activities focused on single/related tasks and less window management.

I strongly believe in virtualized User Interfaces that map to particular tasks that a user is currently doing. However, more investigation is required to really understand the right level of virtualization. It could be at a macro level: entertainment, office work, programming, etc. Or it might be more granular: home financials, music composing, correspondence (e.g., letters, faxes, email, voice, etc.), watching TV, etc. The reason I think this makes so much sense is that it allows us to create a more familiar environment for a user and still let the user switch from task to task. Further, I think this task-based view is much closer to the way average users work (e.g., I need to write some checks and see where the portfolio is today).

Without a doubt, the best computer concept is the concept that you don’t have to invent (because it already exists in the real world).

# Simplicity

We seem to clutter everything. We clutter the desktop. When was the last time you opened the “network neighborhood”? Yet it is on the desktop of probably 100 millions machines. We clutter menus. Look at the cascading menu after menu off the Start/Programs area. We clutter toolbars. And on and on.

We have so many tab dialogs for some functions that we created multi-row-tabbed dialogs. Some of these tab dialogs are amazingly complex. (By the way, the fact that the tabs all change positions when a back-row tab is clicked is extremely confusing and difficult to use.) We should only include a function in the system if a large number of users need the function. Basic functions should be immediately apparent – not masked by reams of options that most customers will never use.

The **Administrative Tools** folder (off the **Programs** **Start** menu) in NT 5 Workstation Beta 1 is a classic example of not designing for the bulk of users. It is more complicated than NTW 4 – which was already much too complicated for the average user. Most users just never need to know about most of these functions. In addition, the current user interface is designed for a computer expert. We have to change our mindset totally here.

We can make systems much smarter than they are today. I think some of the intelligence in the office applications is quite good. There is no equivalent of this in the operating system however. Simple heuristics go a long way. For example, if we can’t sense a network adapter, then we shouldn’t have timeouts and we shouldn’t ask the user anything about a network. We should automatically reconfigure the system. Suppose we do find a network that is operating, but if we can’t find a DHCP server, then we should be able to automatically assign an IP address. If a DHCP server comes back online, then we should automatically switch to the right IP address. If someone is looking at a help topic, then we should be able to step him or her through the tasks to address their question. We should be using as much automatic and dynamic self-configuration as possible. The system should be viewed as a true organism that is constantly adapting to its surroundings (usage, environmental changes, etc.).

Still, simplicity is about more than just being smarter. I expect that most early users believe that Office is quite complicated. Why? Because Office doesn’t ease a user into using the product --- starting with the basics and gradually expanding to richer and richer power. My Mom was overwhelmed when I first introduced her to Office. She now has a limited set of functions that she knows and she tries to avoid deviating from them.

I believe that user interfaces need to be scaled. The scaling can be done in many ways. The most naïve way to do this is to only show the basics and then create an advance area(s) where the “pros” go. Another technique is to have toolbars not show up by default. Another more high-tech approach is to track which features/functions a user does frequently and slowly evolve the interface to favor/highlight these functions. We must really improve in this area in the future.

I hate most wizards. They simply cover up the complexity that exists in the system. If a wizard fails for some unplanned reason, the user is dumped right back in the underlying complex system. If the underlying system weren’t complicated, then probably a wizard wouldn’t have been needed in the first place. A wizard might be useful in certain multi-step interactions with the system, but in general they could be avoided if the system was simpler and smarter.

In short, I think we need ways to scale the UI from simple/basic operation to more advanced operation. We need the system to be more automated about configuration and learning what a user uses and might want to use. And we should hide (or totally remove) clutter that is in everyone’s face unless the bulk of naïve users will use the function.

# Obviousness

If you delete an icon on the desktop, what do you think happens? Does the object really go away? You would think so – it’s the obvious thing – after all you deleted it, right? Well, it depends on the object. In some cases the entire object goes away (e.g., spreadsheet – assuming no links!), but in other cases only portions of the total object goes away (e.g., program executables). Some programs today have icons on the desktop, menu items in the **Start** menu, tons of files somewhere on the disk, and a series of registry entries (sometimes dispersed in several areas of the registry). The obvious thing just doesn’t happen when you delete objects. In fact, our inconsistency concerning deletion is one reason why most user’s disks are full of unnecessary files; the user thinks they were already deleted!

The system should never pretend to do something transparently if it can’t do it perfectly 100% of the time. The cancel button in the browser is a great idea, for example. It is horribly confusing when the system tries to hide something that ends up showing up in certain failure cases. Better to let the user participate in the environment in this case. It is much more obvious this way. The conceptual model then is the same for the user and the computer. Another example deals with Intellimirroring. We should NOT hide the fact that there is a difference between a user who is connected vs. standalone. Trying to pretend like there is no difference will clearly not work and the user will be surprised at random times as the transparency fails.

Tasks that users perform frequently or would likely have to do at some point should be very discoverable. It is critical to understand a prioritized list of tasks and optimize the obviousness of these. The operating system is a mixed bag today. An example of a big improvement is the Small Business Server. But, in general, our usability is very poor across our products. I don’t think we ever generated such a prioritized list for the operating system. If we had, then things like configuring and accessing Mail (or fax) would have been much simpler than it is today. Designers must be very hardcore to ensure that the ease of use for the key tasks is not undermined.

As an example, consider Backup. Where should the program item and/or icon be placed? Should the dialogs give the user lots of power through many choices (incremental, scheduled times, particular files, job definitions, etc.)? The obvious task the user wants to do is to simply backup the system (or restore a file). Any design that doesn’t make this task *extremely* obvious and easy is failing at this design principle.

# Safety

Users should be protected from making errors. This seems obvious enough. We do some of the easy things today (e.g., have a recycle bin, reconfirm a deletion or format, etc.). But, we do virtually nothing to provide any safety for the system as a whole. Users can make a machine non-bootable in less than 10 seconds. They can stop networking from working in 10 seconds. They can break an application in about 10 seconds.

We should never allow a system to be put in an unusable state unless the hardware has failed. Today, we have several problems. First, we let people metaphorically play with the integrated circuits of the software by being able to play with system files loaded on the disk. By system files I mean any component of the system that is used in the standard operation of the system. There is no reason for this. There is no reason anyone except a programmer needs to know anything about these files in the system. They should not be able to be seen, deleted, moved, renamed, or tampered with in any way. Drivers (or other system components) should be signed and a user should explicitly agree to use a driver (or other system component) before they can be installed. There should not be any way to partially remove or install an application.

Second, we need to ensure that manual configuration operations (hopefully over time there should be fewer and fewer of these) cannot put the system in an unusable or non-recoverable state. If a manual configuration could make some part of the system (e.g., hardware device, network connection) stop working, then the user should be strongly warned about the danger of the proposed change. We should attempt to make significant changes a single step UI operation so that we can detect the danger easily. In contrast today, configuring a dial-in line requires multiple steps in different areas of the UI with reboots in between. Obviously mistakes are easy and restoring the system to the way it was before the configuration started in very difficult. A global undo is a good idea. In addition, issues arise when something is reconfigured (e.g., deleting an ISDN board) where we leave partial bits of information around that confuse the system. When something is deleted, *every* trace should be removed from the system. There shouldn’t be any information in the registry, file system, or user interface associated with the deleted entity. In general, reversibility would build confidence for users to try new things since they would know that they could reverse any bad decision they made.

Having an overall default mode for everything in the system is also a good idea. This is a mode that lets the system operate no matter what, albeit perhaps not as customized as the user might want.

# Personalization

Personal computers are about being “personal”. That means it is critical that users be able to customize the interface they use to the computer – just like people customize/decorate their cars, homes, and desks at work. Personalization can help make the system feel comfortable and familiar.

This is an area where we need to do a lot of work – especially when a computer is shared. Today, we confuse per machine and per user state frequently. We need to be able to customize very clearly for each user --- as well as make it easy to reset the environment to an overall default state for a new user (e.g., family member). Further, it is important that this personalization be moveable easily from one computer to another.

# Some Mini Rules to Consider

1. Data integrity or crashes must be ship stoppers.
2. Gracefully support all known failure conditions. Program defensively being prepared for these failure cases. Given the complexity of the system, don’t expect administrators or the end-user to figure out what the problem is. You (the designer) have more knowledge of what to do than anyone else does. The probability of having a human make the situation worse is very high.
3. Test all limit cases – e.g., disk full.
4. Do not permit a user to change anything that could cause the system to crash (e.g., registry). Do not give a user access to internal parts of the system that should be hidden.
5. Do not use computer terms (e.g., DNS, DHCP, FAT32, scandisk, DLL, etc.) when communicating with a user. THINK like someone who has never seen a computer before. Create and use a glossary of terms for the product. Error messages should only use terms from the glossary. (The “repair” person should have some means to get more diagnostic information.)
6. Make concepts shown to the user work consistently in all cases (e.g., **shortcuts**).
7. Avoid configuration if at all possible (e.g., make systems auto configuring by sensing the environment: memory size, network condition, etc.)
8. Do not show error messages to users unless they can specifically do something about the problem. The system should recover from the problem automatically or simply stop working so that a “repair” person can diagnose the problem.
9. Make the 90% case the design center. Ensure that the basic top 10 tasks are obvious.
10. Integrate resolution of a user issue with the Help system.
11. Assume that the network is error prone and transient.
12. Create scaleable User Interfaces where a user can be slowed exposed to more power, but can start their experience in a simple, non-threatening mode.
13. Avoid wizards that simply cover up the complexity of the underlying system.
14. Use concepts from the real world. And match the semantics of the real world as closely as possible. If a new concept is required, then ensure that it is defined and use consistency throughout the product and documentation.
15. Design per user capabilities in the product from the start so that easy personalization is possible.