

What Do Users Want? Engineering Usability into Software

Larry L. Constantine

Professor of Computing Sciences | University of Technology, Sydney
Director of Research & Development | Constantine & Lockwood, Ltd.

Learn more about usage-centered design and essential use cases at
<http://www.forUse.com>.

"What do users want, anyway?" If Freud had been a software developer instead of a psychoanalyst he might have posed this question. Users always seem to be wanting more and developers do not seem to be too good at figuring out what more to give them. It doesn't seem to be enough for us to write reliable code or load our systems with features. So, what do they want?

What users really want is good tools. All software systems, from operating systems and languages to data entry and decision support applications, are just tools. End users want from the tools we engineer for them much the same as what we expect from the tools we use. They want systems that are easy to learn and easy to use and that help them do their work. They want software that doesn't slow them down, that doesn't trick or confuse them, that doesn't make it easier to make mistakes or harder to finish the job.

This is not an easy assignment. You know it's difficult, because major commercial software packages are often riddled with obvious usability defects. Even the biggest software development houses seem to be better at churning out heaps of fast-changing features than delivering straightforward usability.

It is not that the industry thinks usability is unimportant. Ads for new user interface widgets and GUI development tools fill the trade magazines. Rows of books on user interface design line the shelves in the book stores. All the major software houses have elaborate user interface testing programs supported by expensive usability labs. We lavish more and more attention on user interface programming--as much as two-thirds of some project budgets--and still miss the mark by miles. Something is wrong with this picture.

We may have demonstrated our ability to produce fatware, but not our ability to deliver more value. Much of the fine code we crank out under project pressures will go to waste because most users will employ only the smallest fraction of the features we worked so hard to produce. Odds are that many may not even be able to find some functions in the morass of controls and menus and dialogue boxes.

We end up cursed for including features that are useless and leaving out ones that are essential. We are vilified for making simple tasks complicated.

Some of our efforts to improve may only have made matters worse. We have been rushing to deliver the latest 3D look-and-feel only to have ungrateful users complain they can't read our gray-on-gray dialogues. We pepper our toolbars with icons only to be told there are too many and they are hard to decipher. We devise sophisticated software wizards to make complicated interfaces easier to use when we might have made the interfaces simpler in the first place.

The un-pretty truth is usability does not stem from attractive graphics, sophisticated visual controls, or active agents. You do not get usability from 3D look-and-feel, color icons, or floating toolbars. It is not what widgets you use but how you use them and how they work together that counts.

Ultimately, usability comes from the way the architecture of the user interface fits with what users are trying to accomplish. This means that you have to understand the work that users are doing in order for your software to fit with it. It also means that you must engineer the entire architecture of the user interface, its overall organization as well as the structural details and dynamic behavior, to support that work.

To do this, to engineer usability into software, you need tools. You need tools that help you to make sense of what users are doing and what they need to support it, and you need tools that help you to organize the architecture of complex user interfaces without becoming lost in details or minutiae. As is, you are already going flat out just trying to crank out software that hangs together. You need to be able to focus your attention and use your time efficiently, because while the users are complaining, the boss is breathing down your neck to deliver the new app within the allotted time-box. So, you will also need a simple and flexible process within which to organize your work.

This article introduces some tools to build better tools—simple approaches for delivering smaller, simpler systems that better serve the needs of users. This is certainly not the whole story, and

it won't make you into an instant usability expert, but here is a core of leading-edge ideas that could make a real difference on your next project, helping you to

Standard Operating Procedure

User interface standards and usability guidelines are burdened with high expectations in the effort toward better user interfaces. Microsoft, Sun, IBM, and Apple have all published user interface standards, and many companies devise their own in-house manuals to supplement published standards. But how much do these help? Jared Spool of Massachusetts-based User Interface Engineering has found that published industry standards and guidelines only cover about ten percent of the problems arising in actual projects. Customized in-house standards only help with about another ten percent. Even when followed, user interface standards don't answer most of the questions that developers face.

Some standards even work against usability. One company, having learned the statistic that about 1 in 12 males are color blind to some degree, simply decreed that interfaces were not to use any color. This simplistic edict effectively equalized the playing field by handicapping everyone, restricting interfaces to hard-to-read, gray-on-gray uniformity.

Some of the legacies of past mistakes become perpetuated in standards that mandate or encourage ill-conceived controls, such as scroll bars (see "Graphical Navigation" in *Windows Tech Journal*, August 1994). Or consider the ten different ways to close a standard app in Windows 3.x. Consistency with standards can be a good thing, but arbitrary consistency or consistently bad does not make for more usable software.

engineer usability into software from the start rather than leaving it to chance or hoping to add it on through last-minute spit-and-polish.

Usage-Centered Design

Users and user interfaces were not always the problems they are today. In the beginning, there were no users. Only operators and an occasional maniacal programmer ever actually touched a computer, and they flipped switches and watched lights on a console. There was really no interface for users. You had your punched cards or punched tape and you had your printer. You punched your cards or tape and fed them into a reader and you tore sheets from the printer. End users got reports with columns of numbers. The lucky few got them formatted and arranged in a more or less readable sequence.

Technologists tend to be more comfortable with technology than people, so it is not surprising that computing discovered user interfaces before it discovered users. That kept the attention on technical issues like screen painting and field length, data validation and escape keys. It took awhile to fully recognize that users, real people, were sitting on the other side of the user interface, staring at the screens and hitting the function keys. Perhaps guilt-ridden over having ignored or disdained users for so long, the profession passed through a brief fad of “user-friendly” interfaces, insipid interaction that was often only the thinnest of scrimms over the same old intolerant and inflexible programming.

I AM SORRY, Bruce Marby ID 77623901, “August” IS NOT A PROPER NUMERIC VALUE. YOU MUST ENTER THE DATE IN THE CORRECT FORMAT. PLEASE TRY AGAIN.

Save for those earnest developers of Bob-like software, the mainstream moved from the feigned familiarity of user-friendly interfaces to putting users right at the very center of the entire development process. User-centered design was born, eventually to be re-christened “user-centric,” making it sound a lot more sophisticated.

User-centered development was not really such a bad idea but it, too, missed an important point: all software systems are just tools. Since good tools support work, making someone’s job easier, faster, simpler, more flexible, or more fun, what is really important is not building software around users, but around *uses*. It may be nice to get software and applications developers to understand users, but what really matters is understanding what users are doing or trying to do, to understand the intended and necessary usage. Users are not the center of the universe. To design more usable software the most important issue is neither the user nor the user interface, but usage.

Why Ask Why

Usage-centered design is a relatively new perspective on software development that starts with very basic questions: Why? Why is this needed? Why would users interact with this software? What are they trying to accomplish?

Why ask why? Because just asking the question of why someone would use an application helps focus on fundamental issues of what would make the system more usable.

Why does someone use an on-line telephone directory? They want to contact someone whose telephone number they don't know or don't remember. Efficiently locating an entry on the basis of incomplete or inaccurate information is recognized as a key design issue. A good user interface will make that the focus, reducing the number of steps and operations to find the number. Why does a user stick a bankcard in an automatic teller machine? To identify themselves. The magnetically encoded card is a means to an end; other means, such as voice-print

Principal Principles

If you don't know good user interface design when you see it, you won't be able to design more usable software. Get any two programmers looking at the same user interface and you will get an argument. Everybody has an opinion and everybody has personal preferences. But the real issue is what works. In the absence of objective testing we have to rely on general principles of good human-computer interaction. Some lists of user interface design and software usability principles extend to hundreds of pages. Jacob Nielsen reduces it to ten fairly broad heuristics. From our experience working with developers, we find the following basic principles to be easy to learn and apply to actual design decisions. Five of them are sufficiently grandiose as to merit being called rules of usability; these provide a framework and general objectives for good user interface design. The other six principles cover guidelines for more specific aspects of good interface structure.

Keeping such broad guidelines in mind won't guarantee better user interfaces, but making your decisions on the basis of established principles improves the odds. The idea is to make UI design decisions deliberately and consciously, on the basis of one or more recognized principles instead of on opinions and personal preferences. These do not cover everything; art and aesthetics are not even mentioned. The best user interfaces often do have a certain graphical elegance or visual appeal to them. On the other hand, putting aesthetics before essential uses is a common mistake that often leads to pretty interfaces that are hard to use.

First Rule: Access: "Good systems are usable--without help or instruction--by a user having knowledge and experience in the application domain but no experience with the system."

Second Rule: Efficacy: "Good systems do not interfere with or impede efficient use by a skilled user having substantial experience with the system."

Third Rule: Progression: "Good systems facilitate continuous advancement in knowledge, skill, and facility and accommodate progressive change in usage as the user gains experience with the system."

Fourth Rule: Support: "Good systems support the real work that users are trying to accomplish, making it easier, simpler, faster, or more fun."

Fifth Rule: Context: "Good systems are suited to the conditions and environment of the actual operational context within which they are deployed."

Visibility Principle: Keep all needed options and materials for a given task visible without distracting the user with extraneous or redundant information. Instead of WYSIWYG, use WYSIWYN: What-You-See-Is-What-You-Need.

Feedback Principle: Keep users informed of actions or interpretations, changes of state or condition, and errors or exceptions using clear, concise, and unambiguous language familiar to users.

Structure Principle: Organize the user interface purposefully, in meaningful and useful ways that put related things together and separate unrelated things based on clear, consistent models that are apparent and recognizable to users.

Reuse Principle: Reduce the need for users to rethink and remember by reusing internal and external components and behaviors, maintaining consistency with purpose rather than merely arbitrary consistency.

Tolerance Principle: Be flexible and tolerant, preventing errors where possible by tolerating varied inputs and sequences and by interpreting all reasonable actions reasonably; reduce the cost of mistakes and misuse by allowing undoing and redoing.

Simplicity Principle: Make simple, common tasks simple to do, communicating straightforwardly in the user's own language and providing good shortcuts that are meaningfully related to longer procedures.

analysis or infrared scanning might be considered. Why does a user invoke a dialogue to insert a special character in a document? Because the symbol or character they want is not on the keyboard. The dialogue should thus display only the characters of interest to the user, not ones that are already on the keyboard.

Starting out with the purpose of usage and identifying focal uses can help save development time. The user interface architecture becomes organized around the important things. Developers can avoid wasting time fine-tuning features that will have little impact on usability and concentrate attention on what matters most.

Cases of Use

To build systems to support usage, designers need to be able to understand and represent how usage is organized. Essential use case modeling is a technique that Lucy Lockwood and I developed for designing user interfaces based on the structure of usage. It derives from work by Ivar Jacobson on object-oriented software engineering. We extended Jacobson's "use cases" to apply to user interface design by drawing on the concept of essential modeling developed by Steve McMenamin and John Palmer. Essential modeling is a proven approach to systems analysis and design that tries to identify the essential core of a problem apart from unnecessary elaborations or purely technical or artificial constraints. Essential use case modeling helps developers fill in the blanks about what users are trying to do and what they need from a system to support it.

An essential use case is an abstract description of the sequence of interaction between a user and a system. It is cast in the form of a simple narrative dialogue written from the user's perspective, in the user's language. Each essential use case represents one case of use, something that is complete and meaningful to the user. Essential use cases are simplified and generalized down to the essential core of usage, stripped of specific references to physical properties or detailed features of any given user interface. The objective is to represent the essence of what it is the user intends to do or expects to get from a system, the real purpose of interaction. For example, using a recycling machine at a supermarket might be described with the following essential use case:

<u>User Intention</u>	<u>System Responsibility</u>
put in recyclable stuff	accept or reject
	give receipt or rebate for value
take it and leave	

No mention is made of pushing buttons or displaying messages. We haven't specified the shape of the openings or what materials are acceptable for recycling. This form of the narrative focuses on the user's interest in the interaction, namely, getting rid of the junk and getting the rebate.

Essential use cases are something like scenarios, which may be more familiar, and a lot like use cases, on which they are based. The best way to understand the differences between a scenario, a use case, and an essential use case is by example. A scenario is a concrete description of a very specific interaction, but one that is chosen to be typical or representative. For example, consider a scenario for a user calling a computer-mediated help-line.

“Ian Smith calls the TechnoTech Support Line at 4 a.m. and hears the ‘Enter customer ID’ prompt. He then and keys ‘17682002’ on the telephone keypad, after which he hears the menu of TechnoTech product lines.”

No one would propose designing a system that only works for a customer named Ian Smith who works through the night, but it is hard to tell from such a scenario just what is the general case to be designed for and what comprises unnecessary details of the chosen example.

A use case is a generic scenario, describing one kind of interaction with a particular user interface. A use case for the TechnoTech system might be:

“Customer calls the Support Line and hears the log-on prompt. Customer then keys in an ID number and hears a menu.

A use case assumes a particular user interface, in this case a telephone for voice output and numeric input. There are also implicit assumptions about the user interface in this use case. The customers identify themselves by a numeric ID entered on the telephone keypad. Options are presented as a vocal menu.

An essential use case is a generalized, idealized use case representing abstract interaction. For this example it might take this form:

“Invoke help line; identify self; choose help.”

In other words, scenarios, use cases, and essential use cases represent successive levels of abstraction, idealization, and generalization. The essential use case is simplified down to the barest minimum of interaction. If we carefully design a user interface to fit closely with essential use cases, we have the opportunity to build a system that is both smaller and easier to use. In any case, we keep our attention on giving the user what they really need to support their interests.

Real applications may involve any number of essential use cases that are interrelated in various ways. One use case may be a specialization of another. For example, **getting savings balance** might be a specialization or subclass of **querying account** in a remote banking application. Some use cases might depend on others as subordinate processes. In a charting tool, for example, the use case **inserting staff position** might use **labeling** as a subordinate process. Another very useful relationship is extension, an idea introduced by Jacobson. An extension is a use case that represents an optional or exceptional or alternative interaction that may be inserted in the course of another use case. For example, **browsing symbols** might be an extension to **inserting symbol** in a tool for putting special characters into documents. The **browsing symbols** use case is employed whenever the user doesn’t see the needed symbol.

Extensions, specializations, and subordinations allow us to describe the complete structure of usage more compactly because we don’t have to keep rewriting or copying the same interactions. They also help us to understand how use cases are interrelated so we can organize the user interface architecture accordingly. Sometimes in the early stages of a project we don’t know exactly how particular use cases are related, but we recognize them as having something unspecified in common. We would say they have a certain “affinity.” Figure 1 shows how affinity and the other relationships are represented graphically in an essential use case model.

Role Me Over

Essential use cases are often more easily derived from another abstract model that helps shift the focus of our thinking from users to usage. Users interact with a system in various roles. A role is an abstract relationship between user and system. It is a collection of common interests, patterns of behavior, and expectations, as software methodologist Rebecca Wirfs-Brock puts it.

Consider once again an on-line telephone directory application. Many users will use the system with only moderate frequency, typically to find a single phone number. These users interact in what we might call the **Casual-Calling Role**. They often know exactly who they want to reach but occasionally have only a vague general notion or partial information. Another class of users might make heavier use of the system, often contacting a whole string of people belonging to a particular set or group, such as a special interest group, or all the product managers. This **Social-Coordinating Role** might be played by a secretary, a committee chair, or just a programmer trying to organize a volleyball team. Yet another role is the **Directory-Administrating Role**, played by anyone who has occasion to add entries, define or rearrange groups, or change telephone numbers.

Different use cases are needed to support different user roles. For example, to support the **Casual-Calling Role** we might develop a use case for finding the telephone number of someone whose name or location is not known exactly or for certain. We could call this use case **finding what's-her-face**:

<u>User Intention</u>	<u>System Responsibility</u>
here is what I know	show who found
[repeat until happy]	go away

“What I know” might include the first letter of the last name plus the department. Or it could be a guess at the surname spelling and the area code of the telephone number.

This use case might be extended by **dialing number**, another use case that represents an optional alternative.

<u>User Intention</u>	<u>System Responsibility</u>
get me that one	dial number

The advantage of separating this out as an extension is that we can use it to extend other use cases. For example, it is just as useful to extend the use case for looking up someone specific whose name is known exactly or to dial a number to check it when a new entry is created in the **Directory-Administrating Role** role.

In Context

User roles and essential use cases help us to understand what users are trying to do and what the system needs to do for users, but it does not tell us what to put on the user interface and how to organize it. We may sometimes go directly from essential use cases to user interface prototyping, but usually that is a wide gap to bridge in one span.

This is a Test, Take Notes

Usability testing is an important part of building better software. Only objective testing can ultimately resolve some gray areas where it is unclear just what will work and what will not. Well structured testing can also uncover usability problems that no design guidelines cover and no expert inspection could identify.

Usability testing comes in many variations but the theme is simple: You sit users or available substitutes down in front of some version of the software and watch them try to use it. What you give them to work on, what you have them try, what you watch for, and how you analyze the results is what makes it interesting and, in some cases, expensive. The favored scheme is to build a usability testing lab, equip it with an array of computers and audio-video equipment, then staff it with psychologists, technicians, and human-computer interaction specialists. This gives you a visible edifice to point out to visitors and industry analysts to prove your commitment to making software usability a top priority.

Field testing is the poor sister in usability. There is no shining lab to photograph for brochures, and the only absolutely required equipment is a notebook and pencil. On the other hand, low-budget field testing has the advantage that it looks at what people actually do when they are doing real work in an ordinary work setting. Not surprisingly, people tend to think and act differently when removed from their offices and brought into a lab to be videotaped attempting some predefined, make-work task.

Testing won't solve the problem of software usability, because testing comes too late. As they say in Total Quality Management, you can't test your way to quality. It costs too much to build it wrong, then ferret out the problems, then devise a fix or a work-around. It is a lot cheaper to design and build it right in the first place or to catch the problems early. If you don't, even the most massive testing program with hundreds of thousands of beta testers won't find enough of the problems. An embarrassingly large number of the defects that are found through testing will end up unfixed because nobody knows how to fix them or because the problems are too deeply and intimately embedded in the architecture of the software.

You should use usability testing to fine tune critical features or to resolve disputes about alternative approaches, to back up hunches with hard data or to prove out daring new departures, but you shouldn't expect to test your way to truly usable software.

It helps to do some thinking about what must be placed on the user interface to support specific use cases without having to worry too much about appearance or the exact shape or choice of UI widgets. We need an abstract model that makes it easier to explore various ways to organize the user interface architecture without having to build it or design it in detail. This is the content model. (We adapted the idea from the "work environment" models used in Contextual Inquiry, an approach to requirements definition developed by Karen Holtzblatt and Hugh Beyer.)

A content model is an abstract representation of the tools and materials, the controls and the containers, that a user interface will need to present to the user to support some one or more use cases for particular user roles. We use simple shapes—squares and rectangles—to represent these abstract tools. Post-it® notes are good for this purpose because they are easy to move around and they have a generic look to them that keeps us focused on the essence rather than details of the appearance or performance of user interface widgets.

For example, to support the essential use cases of **finding what's-her-face** and **dialing number**, we might start off identifying some of the tools and materials we think would be needed by a user trying to carry out these related use cases:

tools/controls
 number-picker
 dialer
 group-picker
 materials/containers
 name-holder
 found-people-bin

These might end up arranged as in

Figure 2. Note that we've begun to add some ideas about behavior and even some implementation notes. A use context model is something like what is called a "low-fidelity" prototype. It doesn't look much like a real screen layout or dialogue box design, but it has the essential elements needed to support the essential use cases.

Navigating Deep Waters

In more complex designs, we need to consider carefully how the user navigates from screen to screen and dialogue to dialogue when carrying out various use cases. For this we use a navigation map that ties the use case narratives to the sequence of interaction with interaction contexts. A navigation map consists of labeled symbols representing interaction contexts and lines representing transitions among them, which are labeled with the conditions of each transition. For example, the on-line telephone directory might support both personal listings maintained by the user and centralized corporate listings. If the user tries to edit a field of a listing in the corporate database, their authorization to do so needs to be verified. We see this in the navigation map shown in Figure 3.

From Uses to User Interfaces

The goal of all this abstract modeling is to come up with something concrete: a design for a user interface architecture that closely conforms to the essentials of the work users are trying to accomplish. The content model and the navigation map can be used as rough guides to a user interface prototype, but there is typically still a lot of work to do and plenty of room for creative graphics design and clever software engineering.

Figure 4 shows an example of an initial paper prototype, obviously incomplete, for the on-line telephone directory application. Speedy and flexible support for **finding what's-her-face** is achieved in this version by fine tuning to the essential use cases. A row of editable fields populate the data grid below through incremental search as the user types. As soon as the search is sufficiently narrowed by whatever the user enters in the search fields, simply double-clicking on the desired number or selecting the number and clicking on the "Dial" button will dial it.

In the absence of a usage-centered approach that informs the entire process with a focus on purpose and essentials, the resulting screen sketches or paper prototypes can look very reasonable but often turn out to be surprisingly clumsy to use. Designs like the one in Figure 5, using conventional controls and cascaded dialogues, are typical of initial student solutions to this problem and even show up in some commercial products for managing personal information. There is nothing wrong with such an interface until you go to use it.

Although this outline of essential use case modeling may make it seem like a fairly linear and rigid process, in practice we often find ourselves bouncing from pillar to Post-it®. Overall, the process might be represented as in Figure 6. For the telephone directory application, we might start with the idea of a single view into a two-part database with personal listings on each PC and corporate listings on a server. This is really an internal software design decision. We might then have an inspired idea for the icon on the dialing button, a detail of the actual user interface prototype. From there we might move to the essential use case for **dialing number** before considering various user roles. We might work all the way

all through to the use context for **finding what's-her-face** before writing out essential use cases to support the **Directory-Administrating Role**. Only then would we start working out the navigation map.

Essential Reading

If you want to learn more about engineering software to meet the essential needs of your users, check out these sources: Larry Constantine & Lucy Lockwood, *Software for Use*, (Addison-Wesley, 1999) plus their definitive paper, "Structure and style in use cases for user interface design" (<http://www.foruse.com/Files/Papers/structurestyle2.pdf>). A variety of usability issues are covered in Constantine's *The Peopleware Papers* (Prentice Hall, 1995) and Jacob Nielsen's *Usability Engineering* (Academic Press, 1993). For more on use cases, turn to the book by Ivar Jacobson and colleagues, *Object-Oriented Software Engineering* (Addison-Wesley, 1992.) and Ian Graham's, *Migrating to Object Technology* (Addison-Wesley, 1994). The classic source on essential modeling is still Steve McMenamin and John Palmer's, *Essential Systems Analysis* (Prentice Hall, 1984).

In other words, the strategy is really a flexible concurrent modeling process. All the essential models and the user interface prototype are important deliverables, but they need not be developed in strict sequence. The goal is to go with the flow, working in whatever way carries the process efficiently forward toward a simple and robust user interface architecture.

Power to the Programmers

I used to say in lectures that organizations will not be able to produce significantly better software unless they make usability someone's job. I have slowly come to realize that this is not enough. Too many seemingly small design decisions made in the course of programming and analysis turn out to have major consequences for system

usability. Usability has to be part of everyone's job. The usability experts aren't going to solve your problems for you. It's up to you to engineer usability into software.

Learn more about usage-centered design and essential use cases at <http://www.forUse.com>.

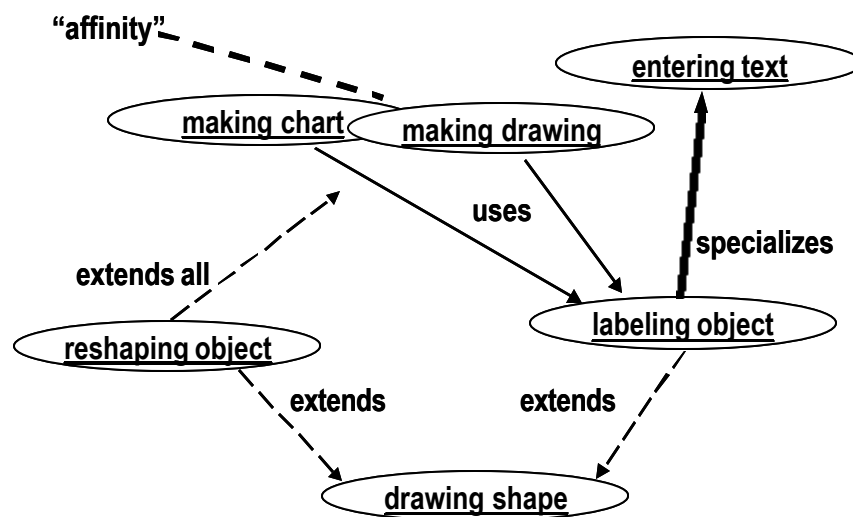


Figure 1 - A concocted example showing how relationships among essential use cases are modeled.

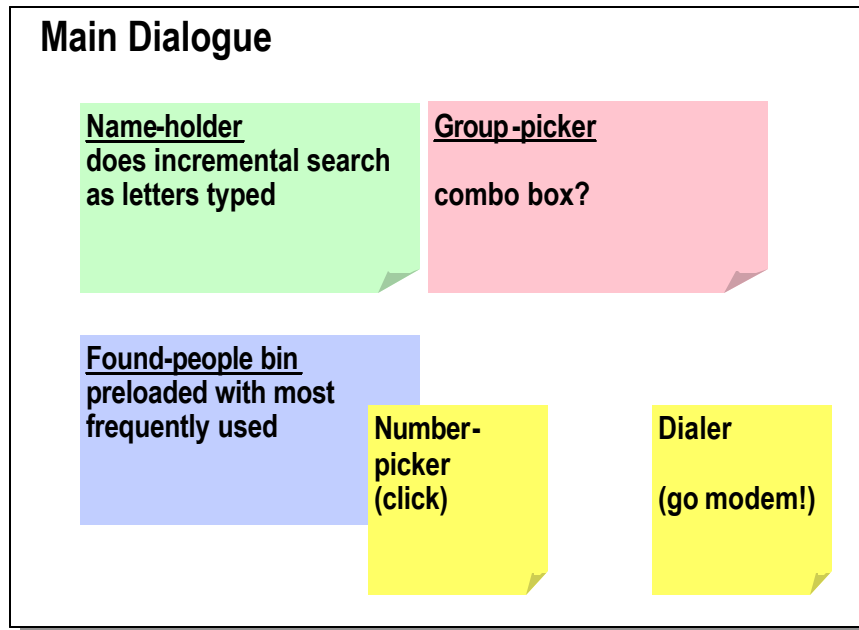


Figure 2 - Preliminary content model for the **finding what's-her-face** and **dialing number** se cases.

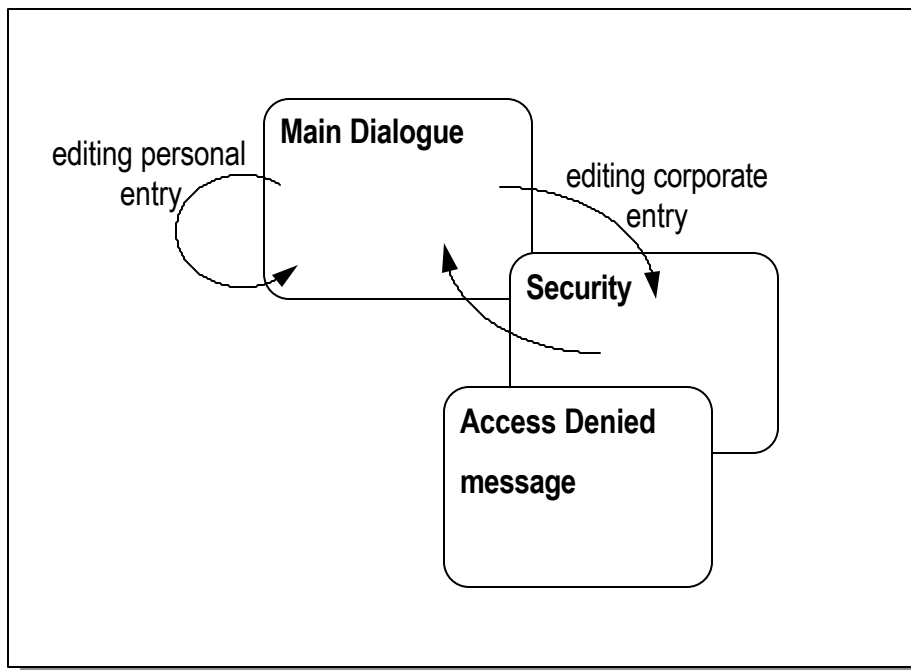


Figure 3 - Partial navigation map for the on-line telephone directory application.

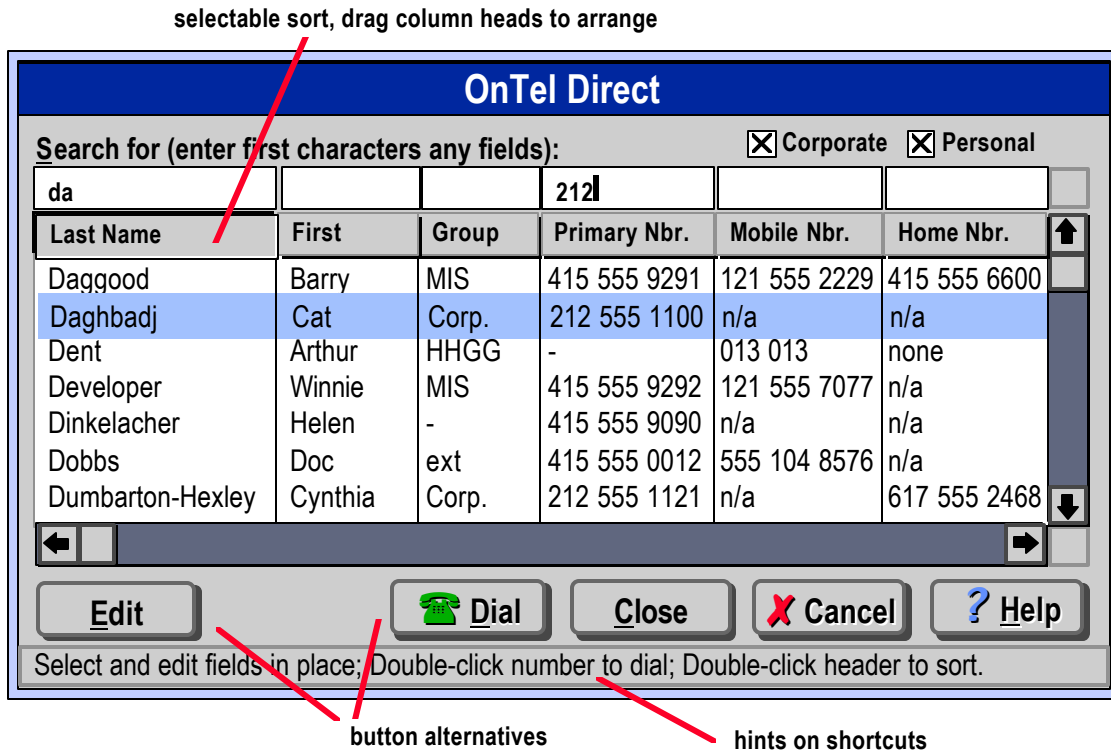


Figure 4 - Fragment of initial paper prototype for the on-line telephone directory application.

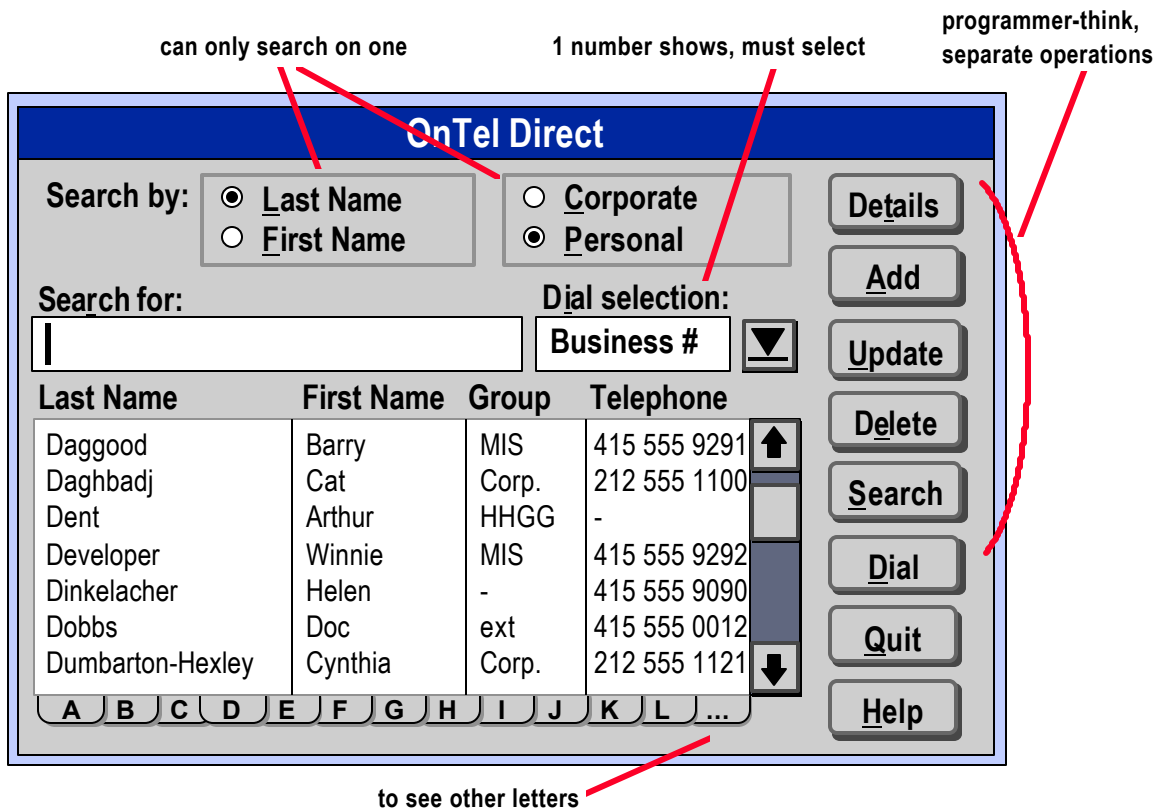


Figure 5 - Example of “conventional” paper prototype design for the on-line telephone directory application.

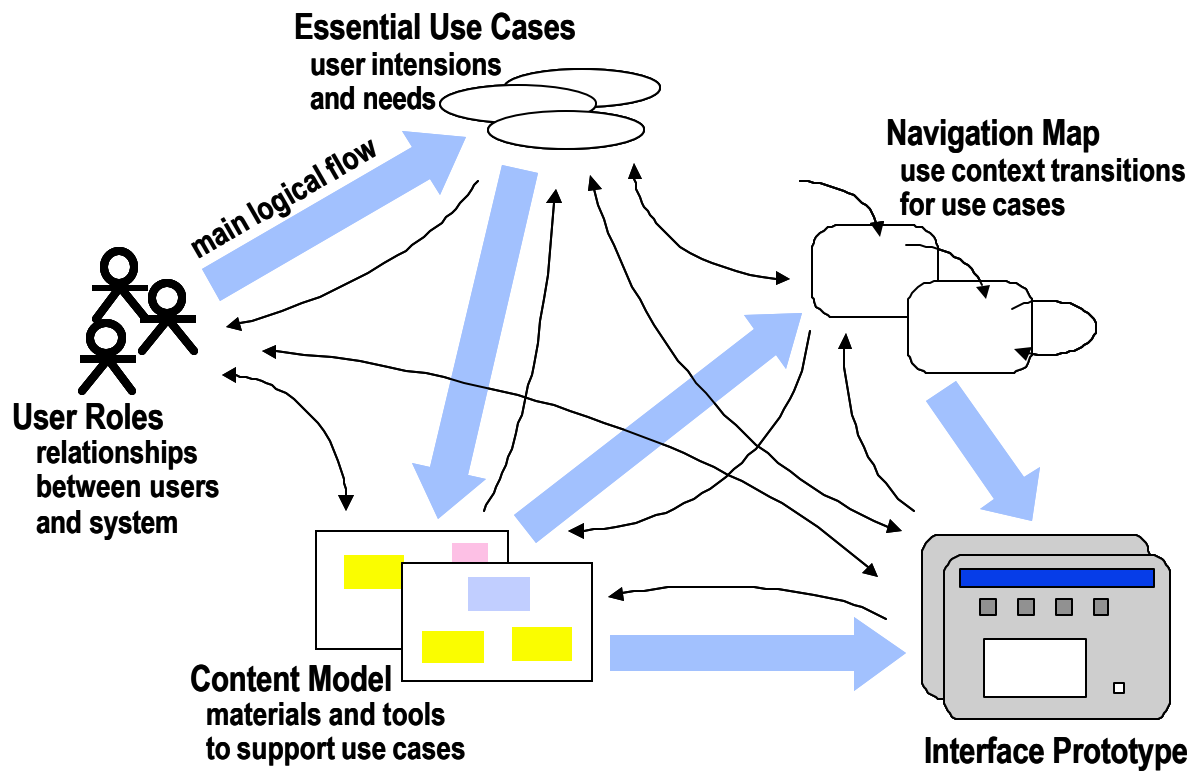


Figure 6 - Overview of a usage-centered concurrent modeling process.