Conceptual models: begin by designing what to design

Article *in* interactions · January 2002

DOI: 10.1145/503355.503366 · Source: DBLP

2 authors:

Jeff A. Johnson
University of San Francisco
116 PUBLICATIONS **1,783** CITATIONS
SEE PROFILE

Austin Henderson
Rivendel Consulting and Design
102 PUBLICATIONS **5,314** CITATIONS
SEE PROFILE

Conceptual Models:
Begin by Designing What to Design

Jeff Johnson, UI Wizards, Inc.

Austin Henderson, Rivendel Consulting & Design, Inc.

Introduction

Suppose you are designing a software product, electronic appliance, or web service. You've gathered functional requirements from Marketing and from prospective customers and users. You've done a task-analysis and created user profiles. What's your next step?

For many designers, especially those new to user-interface design, the next step is to sketch the control panels and dialog boxes of their product or the pages of their web service. Such initial sketches are usually high-level and low-fidelity – showing only gross layout and organization.

If you begin your design phase by sketching, we believe you've missed a step. Sketching amounts to starting to design how the system *presents itself* to users. It is better to start by designing what the system *is* to them. That is, by designing a *conceptual model*.

Let's consider examples of conceptual models. Assume you are designing:

A website. Is the site
a) a collection of linked pages, or
b) a hierarchy of pages with some crosslinks?

Breadcrumbs for website navigation. Do they show
a) the history of pages you have gone through to arrive here, or
b) the place of this page in the hierarchy of pages?

Support for discussion grouped around topics. Is the structure
a) a set of threaded lists one for each subject, or
b) a set of postings each with potentially related subjects?

An application for creating newsletters. Is a newsletter
a) a list of items, or
b) a set of pages each with layout of items?

A platform for creating questionnaires. Is the questionnaire
a) a linear list of questions, or
b) a branching tree of questions?

These decisions matter. Depending on how you choose, users will think of things differently, the objects will be different, the operations users can do on them will be different, and how users work will be different. If you try to avoid choosing, to have it both ways (and, of course, most designs have more than two ways of going), users will get a confused understanding of the system and confused direction on how to think about their work. *Not choosing* is tempting, because these decisions are almost always difficult to make: usually they involve tradeoffs between simplicity and power (tough call!); and they always depend on what the user is doing, which means being clear about the users tasks. But in the end some sort of decision on the conceptual model will be made, even if only as a side-effect (often bent and uncertain) out of the rest of the design process.

Tough decisions, but essential, as we see it. And better done right up front when it is not made even more difficult by being encumbered with lots of dependent details. Our position: Get the bone structure right, then flesh it out.

By carefully crafting an explicit conceptual model focused squarely on the target task-domain, and then, and only then, designing a user interface from *that*, the resulting product or service will be simpler, more coherent, and easier to learn. In contrast, if you jump straight into designing the user interface, you are much more likely to develop a product or service that seems arbitrary, incoherent, and overly complex, not to mention heavily laden with computer-isms. (For an example, see Sidebar 1: A Web App Without A Task-Based Conceptual Model.)

Designers with strong backgrounds in human-computer interaction and user-interface design are probably well-aware of the value of conceptual models. However, our experience with our clients indicates that conceptual models of this sort are almost completely unknown outside of the HCI community, especially among web-designers and software programmers.

What a Conceptual Model Is

A *conceptual model* is a high-level description of how a system is organized and operates. It specifies and describes:

- the major design ***metaphors*** and ***analogies*** employed in the design, if any.
- the ***concepts*** the system exposes to users, including the task-domain data-objects users create and manipulate, their attributes, and the operations that can be performed on them.
- the ***relationships*** between these concepts.
- the ***mappings*** between the concepts and the *task-domain* the system is designed to support.

In using an interactive system (electronic appliance, software program, or web service), reading its documentation, and talking with other people who use it, users construct a model in their minds of the system and how it works. This allows them to predict its behavior and generalize what they learn to new situations. If the designers take the trouble to design and refine a conceptual model for the system before they design a user interface for it, users will be able to more quickly "figure it out". Furthermore, the model they "figure out" will be more like the one the designers intended. A conceptual model of an interactive system is therefore:

- an *idealized view* of the how the system works – the model designers hope users will internalize;
- the *ontological structure* of the system: the objects, their relationships, and control structures;
- the *mechanism* by which users accomplish the tasks the system is intended to support.

For example, suppose you are designing an online library catalog. The conceptual model might include:

- ***metaphors*** and ***analogies***: e.g., the information is organized as in a physical card-catalogue.
- ***concepts***: e.g., item (with attributes: title, ISBN, status; with actions: check-out, check-in, reserve), subtypes of item (e.g., book, periodical issue, LP, video), periodical volume, user account (with attributes: name, items checked out), librarian;
- ***relationships***: e.g., a book is one *type* of item, periodical volumes *contain* issues;
- ***mappings***: e.g., each item in the system corresponds to a physical item in the library;

For an example of a conceptual model for a different task-domain, see Sidebar 2: Managing Checking Accounts: Objects, Attributes, Actions.

Simple: A conceptual model should be as simple as possible while providing the required functionality. An important guideline for designing a conceptual model is: "Less is more." If, for example, you're designing a search facility for the Web, do your intended users really need full boolean search capability? If not—if a simpler search mechanism covers the user's needs—don't burden the design with the more complex capability. Similarly, if you're designing a route-following application, is "turn NNE" needed, or only "turn right"[1]. And beware, simple ain't simple: it often takes a lot of thinking (and testing) to deciding which model will be simplest!

Task-Focused: The more direct the mapping between the system's operation and the task-domain it serves, the greater the chances that the designers' target conceptual model will be correctly reproduced and adopted by the users (Norman, 1986). For example:

> You are designing a software product for creating and managing organization charts. Is an organization chart
>
> > a) a collection of boxes, box labels, box layout, connector lines, and attributes thereof, or
> >
> > b) a collection of organizations, sub-organizations, employees, and attributes thereof?

Model B maps more directly to the users' task-domain, and so will be easier for the users, who presumably already understand organizations, to master. In contrast, Model A focuses on the graphic appearance of an organization chart, rather than on its function.

What a Conceptual Model Is Not

The conceptual model of an interactive system is *not the user interface*. It is not about how the software looks or how it feels. It does not mention keystrokes and mouse-actions, screen graphics and layout, commands, navigation schemes, dialog boxes, controls, data presentation, or error messages. It does not say whether the software is operated through a GUI on a personal computer or by voice-commands over a telephone. It describes only what people can do with the system and what concepts they need to understand to operate it. It refers only to task-domain objects, attributes, and actions.

The conceptual model is *not the users' mental model* of the system. Users' mental models of systems are not accessible to designers in any objective sense. Designers should not waste time trying to determine what the users' "mental models" of the system are (Nardi, 1993). Different users are likely to have different mental models of a given interactive system anyway. Conceptual models are more usefully thought of as a design-tool—a way for designers to straighten out their thinking before they start laying out widgets. It is the designers' responsibility to devise a conceptual model that makes sense to users based on users' understanding of the task domain. In other words, a conceptual model may be the *basis* for users' mental models of the system, but that is not its primary purpose.

The conceptual models are *not use cases* (also known as *task-level scenarios*). Use cases are stories about the domain tasks that users will have to carry out in their work. They are supposed to be expressed in a system-neutral way, so as not to specify the design of the system. Use cases emerge from study and analysis of the task domain – through interviews, ethnographies, focus groups, contextual inquiry, and other methods. They can either be *input* to the design of the conceptual model or they can *emerge* from it (see below); therefore, they are often included in

[1] If this example bothers you because it's comparing apples and oranges—different *ways* of thinking about directions, then good! you are thinking about conceptual models. *[Note: Not sure this footnote adds enough to warrant keeping it.]*

documents about conceptual models. However, a set of use cases is not a conceptual model: use cases focus on *tasks*; the conceptual model focuses on the *system*.

Finally, a conceptual model is *not an implementation architecture*. An implementation architecture contains concepts – objects, attributes, actions, and control structures – that are required to implement the system. Some of these concepts in the implementation architecture may correspond to concepts in the conceptual model (e.g., a BankAccount class vs. the concept of a bank account), but if so, one is a technical object while the other is an abstract construct. Of course, an implementation architecture will also include implementation objects that are of no concern to users (e.g., streams to the file system), which should have no place in the conceptual model[2].

Object/Action Analysis

An important component of a conceptual model is an Objects/Actions analysis: an enumeration of all the concepts in the model—all the user-understood objects in the system, user-understood attributes of those objects, and the actions that users can perform on each of those objects (Johnson *et. al.*, 1989; Card, 1996). The Objects/Actions analysis, therefore, is a *declaration* of the concepts that are exposed to users. Follow this rule: "If it isn't in the conceptual model, the system should not require users to be aware of it."

Because computer-based systems often provide new capabilities, concepts not found in the task domain – especially a pre-computerized one – often creep into the conceptual model. For example, hardcopy documents in a physical filing system can only be organized one way, but files in an electronic document system can easily be organized in multiple ways simultaneously.

However, each new concept comes at a high cost, for two reasons:

- It adds a concept that users who knows the task domain will not recognize and therefore must learn.
- It potentially interacts every other concept in the system. As concepts are added to a system, the complexity of the system rises not linearly, but exponentially!

Therefore, additional concepts should be strongly resisted, and admitted into the conceptual design only when they provide high benefit and their cost can be minimized through good user interface design (see the discussion of Quicken™ in Sidebar 2: Managing Checking Accounts: Objects, Attributes, Actions). Remember: Less is more!

Relationships Between Concepts

Enumerating the objects and actions of the task-domain allows designers to notice actions that are shared among objects. Designers can then use the same user interface for actions across a variety of objects. For example, consider a drawing application that allows users to manipulate both rectangles and ellipses. If creation works the same way for both types of objects, when a user knows how to create a rectangle and wants to create an oval, - they already know how to do it. Similarly, if users can constrain rectangles to be squares they should also be able to constrain ellipses to be circles. This makes for a conceptual model that has fewer distinct concepts, is simpler and more coherent, and is more easily mastered.

[2] Sometimes this distinction is made by saying that only "user-visible" objects and relations should be in the concepts of the conceptual model. However, this can be confusing, because it tends to direct our thinking toward presentations—how things look—which as we have said is the subject matter of the interface, not the conceptual model. We find "user-understandable" to be a safer term as it directs attention to the users' understanding independent of how things look.

If objects in a task-domain share actions, they can probably be organized in a *specialization* or *type hierarchy*, in which certain conceptual objects are *specializations* of others. If so, making that hierarchy explicit in the conceptual model may help users comprehend it more easily. While only programmers understand object-oriented analysis, most users can understand the idea of specialization: for example, a *checking account* is a type of *bank account*, and a *book* is one type of *product* or *item* a store might sell.

Depending on the application, objects may also be related by a *containment hierarchy*, in which some objects can *contain* other objects. For example, an email folder *contains* email messages, and an organization can *contain* employees.

Finally, concepts in a task-domain are related to each other in *importance*. Some concepts are encountered by users more frequently than others. For example, closing a checking account is an infrequent operation compared to, say, entering a transaction into an account. The relative importance can be used to focus the design: it is more important to make frequent operations easy, even at the expense of less-frequent ones.

From Conceptual Model to Completed Project

Developing a conceptual model as the first design step provides several benefits in later steps:

Lexicon. Once the development team assigns names to the objects, actions, and attributes enumerated in the conceptual model, they have a *lexicon* of terms to be used in the application and its documentation. As the interface is developed, the software coded, and the documentation written, the lexicon can be consulted to ensure that terms are used consistently throughout.

Although the entire team develops the lexicon, it is best managed and enforced by the team's technical writer. This lexicon-manager—whoever gets the job—should constantly be on the lookout for inconsistencies in what things are called. For example: "Yo, Bill. We called this thing a 'cell' in this dialog box, but we call it a 'container' in this other dialog box. Our official name for them is 'cells', so we need to fix that inconsistency.". Software developed without a lexicon often suffers from two common user interface "bloopers": 1) multiple terms for a given concept, and 2) the same term for multiple distinct concepts (Johnson, 2000).

It is also the lexicon-manager's role to be on the lookout for user-visible concepts in the interface, software or documentation that aren't in the lexicon, and to resist them. For example: "Hey Sue, I see that this window refers to a 'hyper-connector.' That isn't in our conceptual model or lexicon. Is it just the wrong name for something we already have in our conceptual model, or is it something new? If it's something new, can we get rid of it, or do we really, *really* need it?"

Task scenarios or use-cases. A conceptual model allows the development team to write scenarios of the product in use, at a level of description that matches the target task-domain. Such scenarios are often called *use-cases*. They are useful in checking the soundness of the design. They can be used in product documentation, in product functional reviews, and as scripts for usability tests. They also provide the basis for more detailed scenarios written at the level-of-detail of the eventual interface design.

Once a conceptual model has been crafted, one can write use-cases or task-scenarios depicting people using the application, using only terminology from the conceptual model. In the case of the checkbook application, for example, it should be possible to write scenarios such as:

> John uses the program to check his checking account balance. He then deposits a check in his account and transfers funds into the account from his savings account.

Note that this scenario refers to task-domain objects and actions only, not to specifics of any user interface. The scenario does not say whether John is interacting with a GUI on a personal computer or a voice-controlled interface over a telephone.

User-interface. A conceptual model gives the designer a clear target for what the interface has to deliver to the user: the look and feel of the objects and actions have to be created, the relationships embodied in the design. The conceptual model then offers the basis for tests of how well the user interface works: can the users manipulate the objects through their representations as the designer intended. (Note: It is tempting to think that the user can tell you about the conceptual model of the system that they have formed in these tests. Resist it! That is setting the bar way too high, and for no reason: It is not at all necessary for successful use of most systems for users either to have the conceptual model "right", or to be able to talk clearly about it. Doing does not require talking!)

The *user interface design* translates the abstract concepts of the conceptual model into concrete presentations, controls, and user-actions. The user interface should be designed *after* the conceptual model has been designed. Task-scenarios can then be rewritten at the level of the user-interface design, for example:

> John double-clicks on the icon for his account to open it. A separate window opens showing the current balance. He then clicks in the blank entry field below the last recorded entry and enters the name and amount of a check he recently received. ...

Implementation. Readers who are programmers will have noticed the similarity between the object/action analysis described here and the object-oriented analysis that is a common early step in software engineering. Although object/action analysis is restricted to *user-understood* concepts while object-oriented analysis is not, having done an object/actions analysis provides a first cut at the object-oriented analysis. Therefore, developing a conceptual model is not a simple added cost for a project; it produces outputs that save costs in the software development stage.

Documentation. A conceptual model provides the documentation team with the material that they will have to provide to the user to help with learning the system (help material, documentation). A clearly defined conceptual model is a good place to start, and should be coupled at all points with the descriptions of tasks and interface actions (see preceding).

Design process. Because almost everyone on the development team is orienting to the conceptual model, the conceptual model can also a central coordination point for members of the team as they design and develop the system.

The centrality of the conceptual model and its potential role in orchestrating the design process has one very strong implication for design activities and their relationship with the conceptual model:

- Unilateral additions of concepts to the conceptual model by any team member is not allowed.

For example, if a programmer thinks a new concept needs to be added to the software, she must first persuade the team to add the concept to the conceptual model; only then should it appear in the software. Or again, if a documenter finds that they have to introduce an additional concept to explain the system, that change must be reflected first in the conceptual model (with the whole team's agreement), and then it will appear in the documentation.

The process will usually not be linear. As design proceeds from conceptual model to user interface to implementation, it is most likely that these downstream designs will reveal problems in the conceptual model. (It is tough to get it right the first – or even the fifth – time!) Early usability testing can, and should, be designed to accelerate this process: low fidelity, quick prototypes can be focused on the important parts of, and questions in, the conceptual model. Lightweight usability testing can thus evaluate the conceptual model as well as the UI design.

If testing exposes problems in the conceptual model, go back and change it. Resist the temptation to treat the conceptual model as "dead" after an initial UI has been designed from it. If you don't keep the conceptual model current as you improve the design, you will regret it in the end, when you have no single coherent high-level description on which to base user documentation, training, or later system enhancements.

Of course, changing the conceptual model is painful: it affects the user interface, the documentation, and the implementation. The entire team is affected. But the conceptual model is the single most important part of your design. Therefore, it pays to make it as simple and task-oriented as you can, then do whatever you need to do to reconcile the rest of the design with it. Otherwise, your poor users will have little chance of understanding the user interface, because it will be based on a muddled conceptual model.

Conclusion

Good user interfaces start with clean, simple, task-oriented conceptual models. The conceptual model is the bones of the design. One nice thing about this is that the conceptual model is much smaller than the whole design. It is something that can be held in mind and worked on. Get the conceptual model in hand before adding all the complexity of everything else.

Once you have the conceptual design, all the other design and implementation activities can and should be grounded in it, feeding it further (task scenarios, evaluation), building on it (user interface, lexicon, implementation, documentation, evaluation). Because the conceptual model is so central, it is important to ensure that everyone agrees on it. Because changes that affect the conceptual model affect everyone, all changes must be made jointly. The conceptual model is the central point of discussion and site of debate.

So at the outset, and throughout, let the sketching follow the modeling. Before you design, design what you are designing: design a conceptual model.

References

Card, S. (1996). "Pioneers and Settlers: Methods Used in Successful User Interface Design", in M. Rudisill, C. Lewis, P. Polson, T. McKay (eds.), *Human-Computer Interface Design: Success Cases, Emerging Methods, Real-World Context*, Morgan Kaufmann.

Johnson, J., Roberts, T., *et. al*. (1989). "The Xerox Star: A Retrospective", *IEEE Computer*, September.

Johnson, J. (2000). *GUI Bloopers: Don'ts and Dos for Software Developers and Web Designers*, Morgan Kaufmann.

Nardi, B. and Zarmer, C. (1993). Beyond models and metaphors: Visual formalisms in user interface design. Journal of Visual Languages and Computing 4, 5-33.

Norman, D.A. (1986). "Cognitive Engineering", in D. Norman and S.W. Draper (eds.), *User-Centered System Design*, Lawrence Erlbaum Associates.

================ Side bar 1 ====================

A Web App Without A Task-Based Conceptual Model

A large database company has a website that its external consultants use to log hours worked. Is its user interface based on a task-focussed conceptual model? You be the judge:

- To log a week's worth of hours, consultants click on "Create Record". Why "Create Record", rather than, say "Log Hours" or "Log New Week"? Because the information is being stored in a database, so a new database record must be created in which to store the new data.
- If a user succeeds in logging a week's hours, the system displays the message: "Success: new row inserted." Huh? Not only does this message seem unrelated to logging hours, it seems unrelated even to the software's own term for the function: "Create Record".
- If a consultant forgets that already she logged her hours for a particular week and tries to log the same week again, the system displays the error message: "ORA-00001: unique constraint (CLEATS.PA_REPORT_HEADERS_U!) violated", informing the user that some internal software constraint has been violated rather than that, e.g., "Hours for that week have already been logged".
- The function for changing one's password accepts any character sequence as a new password, even though the Login function won't accept non-numeric passwords. Thus, it's possible to set your password to a string that the Login function flags as an entry error. You then cannot login.

================ End of sidebar 1 ====================

=================Side bar 2 ==============

Managing Checking Accounts: Objects, Attributes, Actions

If we were designing software for the task-domain of managing checking accounts, the object/actions analysis would, if properly task-based, include objects like *transaction*, *check*, and *account*. It would <u>exclude</u> non-task-related objects like *buffer*, *dialog box*, *mode*, *database*, *table*, and *string*.

Regarding attributes, it would make sense in a task-based conceptual model for **checks** to have a *name*, a *number*, and a *date*; for **accounts** to have an *owner* and a *balance*; and for **transactions** to have an *amount* and a *date*. However, a conceptual model in which **transactions** had a *byte-size* or an *export encoding* as user-visible attributes would not be task-focused and would detract from the learnability and usability of the software, no matter how much effort went into designing the user interface.

Finally, a task-based conceptual model would include actions like *writing* and *voiding* checks, *depositing* and *withdrawing* funds, and *balancing* accounts, while excluding non-task-related actions like *clicking* buttons, *loading* databases, *editing* table rows, *flushing* buffers, and *switching* modes.

A checking account management application probably has to support recurring transactions, such as paying the electric bill each month. Therefore, it may seem necessary to include objects like *transaction templates* and actions for *defining* and managing templates.

But consider how repeating transactions are handled in Quicken™, a checkbook management product from Intuit. Quicken's designers recognized that entering recurring transactions should be very easy. The designers could have fulfilled this need by including an explicit template-management facility, with commands like "Define Template" and "Use Template". Wisely, they didn't do that. It would have added greatly to Quicken's overall complexity. Instead, they allowed users to simply record a transaction as if it were a one-time event, then tell Quicken they want to reuse it. Quicken creates a template from the transaction (based on domain-specific rules) and puts it into a list. Users simply click on a listed transaction to reuse it. In this way, the designers of Quicken added the functionality of transaction templates without the excess conceptual baggage that many software applications that offer templates have.

================= End of sidebar 2 =====================