Virtual Windows: Linking User Tasks, Data Models, and Interface Design

Soren Lauesen and Morten Borup Harning, *Copenhagen Business School*

User interface design comprises three major activities: organizing data into a set of windows or frames, defining functions that let the user control the system, and designing the graphical appearance of windows and functions. These design activities can build on analysis results such as task analysis and data modeling, and they can include checking activities such as reviews and usability tests. The goal is to create a system that is easy to learn, is easy to understand, and supports user tasks efficiently. (Some systems are for entertainment rather than task support, and in such cases, task efficiency is of no concern. We will not consider such systems in this article.)

The authors show an approach for designing user interfaces that balances a good overview of data with efficient task support, and allows user validation much earlier than do traditional usability tests.

Many developers wonder whether there is a systematic way to design a user interface. How do we get from data models and use cases to the actual interface? Two systematic approaches have been around for a long time in various versions: the *data-oriented approach* (easy to do with popular database tools) and the *task-oriented approach* (dominant in the field of human–computer interaction). Each approach has its strengths and weaknesses. In this article, we show a new approach called *virtual windows* that eliminates many weaknesses of these two classic approaches.

Two traditional approaches

The *data-oriented approach* starts with a description of the data the system must maintain, typically in the form of a data model (a static class model). From this, designers define a set of windows such that all data is visible.[1,2] The functions tend to be standard functions for creating, updating, and deleting data. The graphical design is dominated by the built-in presentations the database tool offers. Examples of this approach are simple applications made with Microsoft Access or Oracle Forms.

This approach's main problem is that it doesn't ensure efficient task support. We have seen many data-oriented interfaces where the user cannot get an overview of the data necessary for important tasks.[3]

The *task-oriented approach* starts with a

0740-7459/00/$10.00 © 2001 IEEE

list of user tasks (use cases) that the system must support. Analysts break down each task into a series of steps. From there, designers define a window for each step.[4,5] In an extreme version, each window holds only the input and output fields strictly necessary to carry out the step. The user can move to the next window and usually to the previous one as well. Wizards (which are popular in many modern interfaces) follow this extreme approach.

This approach's main problem is that the user never gets an overview of the data available. He or she only sees the data through a "soda straw." Furthermore, real-life tasks are usually much more varied and complex than the analyst assumes. Consequently, the system supports only the most straightforward tasks and not the variants. In some systems, only straightforward tasks must be supported, which is when the task-oriented approach is excellent. Withdrawing cash from an ATM is a good example. In more complex systems, however, another approach is needed.

Why it works

The virtual-windows technique is an approach based on employing data and tasks at the same time. Part of the approach is to design and test the graphical appearance before the functions are defined.

A virtual window is a picture on an idealized screen. With a PC application, think of it as a GUI window on a large physical screen. This window shows data but has no buttons, menus, or other functions. For devices with specialized displays, think of the virtual window as a picture on a large display; for Web systems, think of it as a page or frame. A complex application needs several virtual windows.

The basic idea when composing a set of virtual windows is this: Create as few virtual windows as possible while ensuring that all data is visible somewhere and that important tasks need only a few windows. Furthermore, design the graphical appearance of the windows such that real-life data will be shown conveniently and that users can understand what the windows show. An important part of the graphical design is to decide whether data appears as text, curves, dials, pictures, and so on.

Later, the designer develops the user interface: He or she organizes the virtual windows into physical windows or screens; adds buttons, menus, and other functions; and adds messages, help, and so on.

We have used the virtual-windows approach for several years in teaching and projects such as special devices and business, Web, and client-server systems. Compared to user interfaces designed traditionally, the ones based on virtual windows appear to have several advantages:

- there are fewer windows,
- there is efficient task support (also for task variants),
- users can validate the database, and
- users better understand the final system.

Virtual windows give *fewer windows* than data-oriented approaches because we do not create a window for each entity—the windows group information according to user needs. They also give fewer windows than in task-oriented approaches because we can reuse windows across several tasks.

Not surprisingly, the virtual-windows approach provides more efficient *task support* than the data-oriented approach because the windows are designed for the tasks. Blindly applied, the virtual windows might support straightforward tasks a bit less efficiently than the task-oriented approach, but they can better handle task variants. Less blindly applied, the virtual windows let the designer support important tasks as efficiently as the task-oriented approach.

The *validation* of database contents is the result of the early graphical design with realistic data. Furthermore, all data is shown in a work context, resulting in the user relating to the data more vividly.

Users better *understand* the final system because fewer windows reduce the user's mental load.

In practice, we also test if users understand the virtual windows. They often don't understand the first version, but because the windows are made early—much earlier than paper mockups of the user interface—we can afford to make radical design changes. Because the later physical windows resemble the virtual windows, chances are higher that users will also understand the final interface.

Understandable windows allow the users to form a better mental model of not only the data in the system, but also the system's

The virtual-windows technique is an approach based on employing data and tasks at the same time.

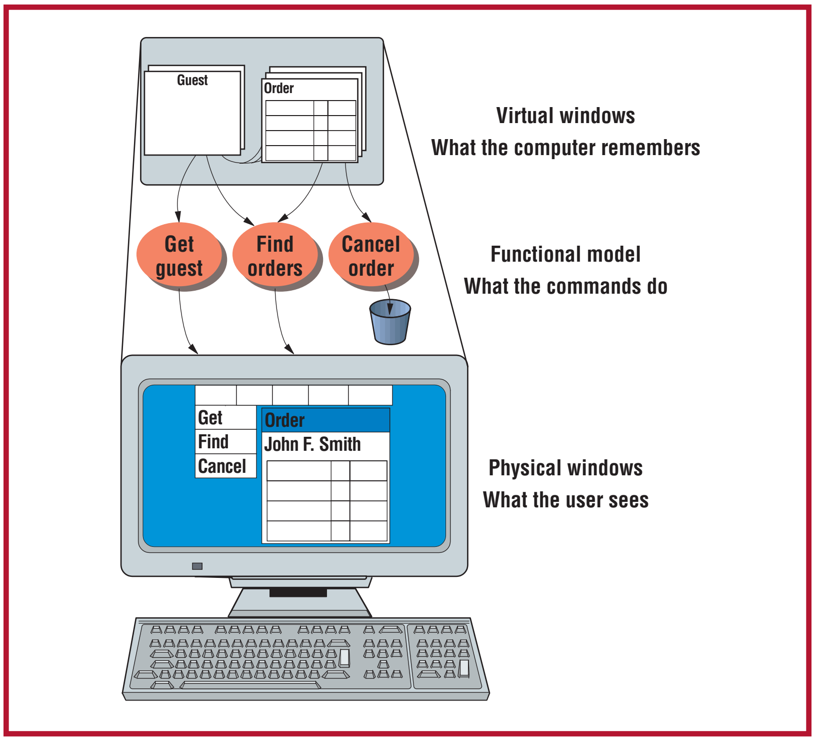


Figure 1. The user forms a mental model of data and functions in the system from what he or she sees on the screen. The virtual windows are a consistent basis for a mental model.

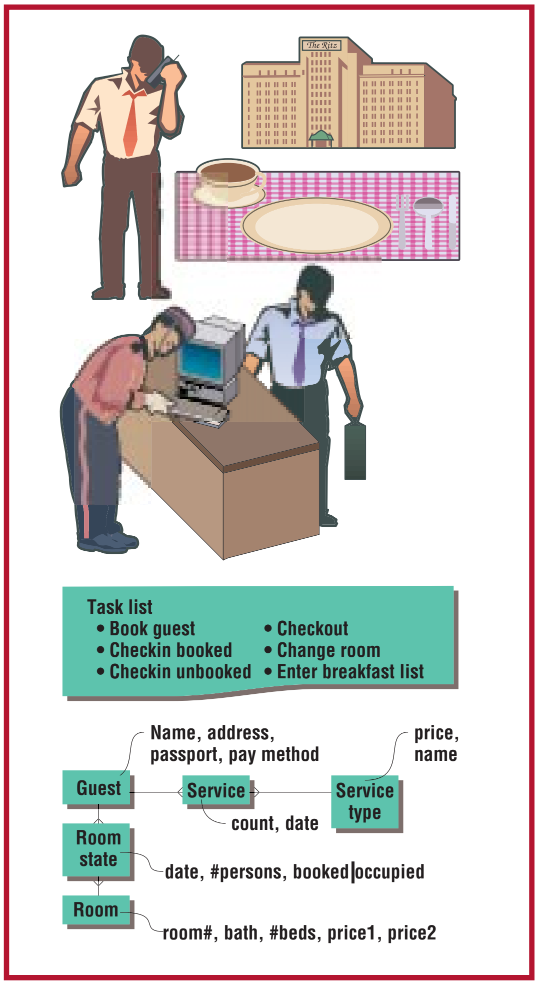


Figure 2. The hotel domain and its formalization as a data model and a list of user tasks.

functions. Figure 1 illustrates the principle. The user sees or enters some data on the screen. When the system removes the window or changes to another window, the user does not assume that the data disappears but that it is stored somehow. The figure shows this stored data at the back of the system. The user thus forms a mental model of what is stored and how it relates to other stored data, based on the way the data appears on the screen. This psychological mechanism corresponds to Piaget's *law of object constancy*: When we see an object and it becomes hidden, we automatically assume that it still exists somewhere.

The figure also suggests that the user understands the functions in terms of the stored data and its relation to what is on the screen. Many users comment that the picture reflects quite well how they imagine the system.

How to do it

Let's examine how to use virtual windows to design a user interface for a hotel booking system.

Make a task list (use cases)

The first step is to list the tasks (use cases) the system must support. This step is part of many analysis approaches and not something special for the virtual-window approach.[4–6]

Figure 2 shows a simple task list for the hotel example. The list shows that our system supports booking, checking in and out, keeping track of breakfast servings, and so on. Implicitly, the list delimits the system's scope. Our list shows, for instance, that the simple hotel system does not support areas such as accounting, personnel, and purchases.

Make a data model

In most cases, the virtual-window approach benefits from a traditional data model (static object model), which is part of many analysis approaches.[7,8] The data model specifies the data to be stored long-term in the computer. Usually, short-term data such as search criteria are not specified in the data model.

A data model is an excellent tool for developers but is hard to understand for even expert users. Figure 2 shows the data model for the hotel example. The data model shows that the hotel system must keep track of guests (one record per guest). To shorten the example, we simplified the system a bit and assumed that each guest only had one stay at the hotel—the system doesn't track regular customers.

The system must also record various services to be paid by the guests such as breakfast servings. There are several types of service. According to the data model, each guest might receive several services, and each service is of exactly one ServiceType.

The system must keep track of rooms. Each room has a list of RoomStates, one for each date in the period of interest. The room's state at any given day can be free or booked. A guest's stay relates to one or more RoomStates corresponding to the dates where he or she has booked a room or actually occupies it.

In most hotel systems, rooms are booked by room type (single or double) while check-in is by room number. To simplify the example, we assume that rooms are booked by room number, too.

Outline the virtual windows

In this step, we outline virtual windows, looking at the tasks one by one. For each task we identify the data to be seen by the user and group them into a few virtual windows. The important trick, however, is to reuse or expand previous windows rather than design new ones for each task.

The step uses a few guidelines. To explain them, we distinguish between a *window type* and a *window instance*. Think of the window type as a template and the window instance as a window filled in with data. As an example, a guest window with data for a specific guest is a window instance, while all the guest windows use the same window type. Here are those guidelines:

- *Few window types*. Keep the total number of window templates small. (It is easier to grasp fewer windows as a mental model.)
- *Few window instances per task*. For each task, the user should access few window instances. (This improves task support.)
- *Data in one window only*. Avoid having the user enter the same data item through several window instances. Preferably, each data item should also be shown in only one window instance. (Seeing the same data item in several windows makes the mental model more complex. Being able to enter it at several places causes confusion about the entry's effect.)
- *Virtual windows close to final physical windows*. Although we assume a large physical screen, the virtual windows shouldn't be too far from what is realistic in the final interface. (Otherwise, the user will not generate a mental model close to the virtual windows.)

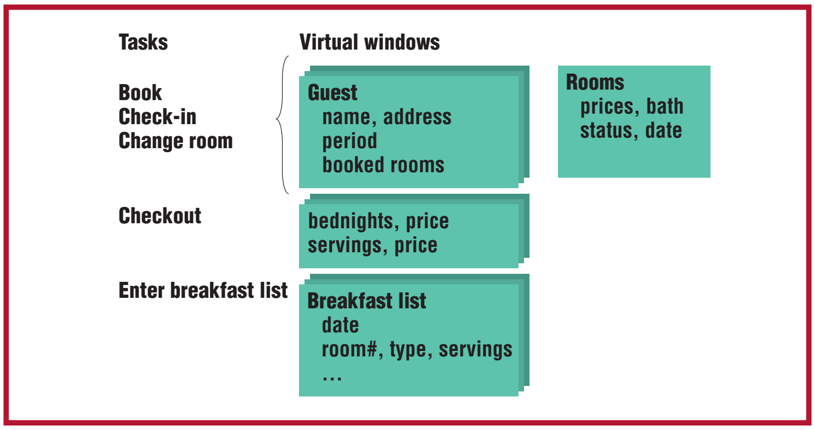


Figure 3. Virtual windows in outline version. A few windows cover all data, and each task needs only a few windows.

These are not rules set in stone. In many cases they conflict, forcing us to strike a balance between task efficiency and ease of understanding (we show examples later). The rules cover only the high-level composition of screens. They do not pretend to cover graphical design for the individual windows.

Windows for booking. We will use these guidelines for the hotel system. Let's start with a frequent task, the booking task. Which data should the user see to book a guest? He or she must see which rooms are vacant in the period concerned, what their prices are, and so forth. The user also must record the booking and the name, address, and other guest information.

Figure 3 shows how this data could be allocated to two virtual windows, Guest and Rooms. The pile of completed Guest windows suggests that we have recorded several guests. We have only one Rooms window, because all rooms are shown there with their occupation status for a period of days. This is also where the user selects free rooms for a guest.

Why do we need two virtual windows? Doesn't it violate the principle of few window types? Apparently, yes, but if we had only one

It is important to test whether users understand the suggested solution properly.

type, the room occupation status would appear in all the guest windows, violating the "data in one window only" guideline. This kind of conflict is common and is not a result of the virtual-window approach—the approach just reveals the conflicting demands.

Although we need two virtual windows for the booking task, we might later choose to show them together on the same physical screen—for instance, as two physical windows or two frames in a single physical page. This is a matter of detailed dialog design, to be handled later.

Check-in, change room. Next, let's consider check-in. Fortunately, the same two virtual windows suffice. If the guest hasn't booked in advance, the procedure is much the same as for booking, except that the room becomes occupied rather than booked. If the guest has booked, we just need to find the guest in the pile of guest forms. We need some search criteria to support that, but again, we delay that to detailed dialog design.

Room changes use the same two windows: one for the customer who wants to change and one to see free rooms.

Checkout. When checking out, the receptionist needs some more data. He or she must see how many nights the guest stayed, what services he or she received, the prices, and the total. It is useful to verify the data with the guest before printing the bill.

Where do we put these data? In a new virtual window? No, because that would violate the "few window types" guideline. Instead, we extend the guest window as in Figure 3. Whether we want to always show the extensions on the physical window is a matter of later dialog design. But when we show them, the graphical look should clearly indicate that this is an extension of the guest data—for instance, it might be shown in the same frame as the other guest data.

Record services. The last task on our list is recording services such as breakfast servings. In principle, we don't need a new virtual window for this—the receptionist could simply find the guest window and somehow record the service there.

However, in many hotels, waiters record breakfast servings on a list with preprinted room numbers. The waiter brings it to the receptionist, who enters the data. The system could handle that with one window instance per room, using a special function that scans through guests in room order. However, this would violate the "few window instances per task" guideline.

Figure 3 shows another solution, a virtual window that holds a breakfast list. This solution is important for task support, although it violates the "data in one window only" guideline, because breakfast servings are also shown in the guest windows. The system is now conceptually more complex because the user has to understand the relation between the guest and breakfast windows. For instance, the user might worry whether the system keeps both of them updated or whether he or she has to do something to have the latest breakfast list transferred to guest windows.

This is a quite serious design conflict between task efficiency and ease of understanding. It is important to test whether users understand the suggested solution properly and, if necessary, to supply adequate guidance in the windows.

Supplementary techniques. In larger systems, defining windows based only on the guidelines listed earlier can be difficult. We use supplementary techniques such as defining larger chunks of data than the simple objects in the data model,[9] setting up a matrix with the relation between tasks and the necessary data for the task,[10] and defining several virtual-window models by using different tasks as the starting point.

Detail the graphics and populate windows

In this step, we make a detailed graphical design of the virtual windows and fill in the windows with realistic data. Still, we don't add dialog functions such as push buttons or menus—that is left to dialog design.

Many windows are rather straightforward to design, and standard GUI controls suffice. This is the case with the guest window and the breakfast list.

Other windows must give a good overview of a lot of complex data, and they are difficult to design. This is the case with the room window—it's easy to show the RoomStates as a simple list of records, but that doesn't give the necessary overview. Our solution uses a spreadsheet-like display with mnemonic codes for room occupation as in Figure 4. It is somewhat complex to implement on most GUI platforms.

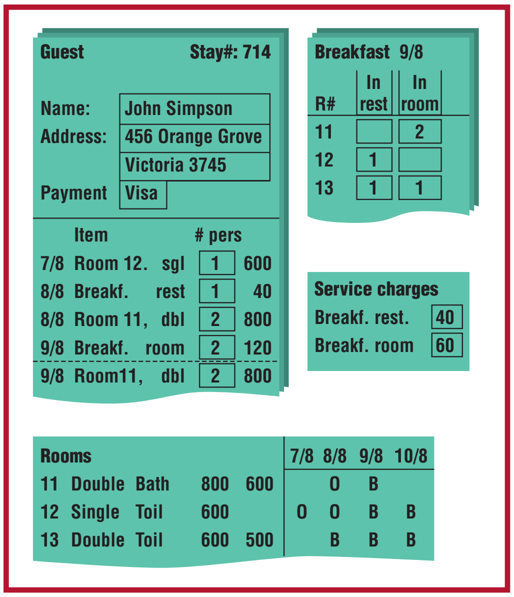
Showing complex data comprehensively is a relatively unexplored area.[11,12] However, virtual windows are excellent for experimenting with advanced presentation forms.

Show realistic and extreme data. It is very important to test the design by filling in the windows with realistic data, as in Figure 4 (abbreviated here for space reasons). We also show a complex, but slightly unusual, situation: a single guest checks into a room and later becomes two guests, who move into a double room. The virtual window shows that, conceptually, this is one stay and one guest.

Apart from filling the windows with ordinary data, it is useful to try filling them with extreme, but realistic, data, which often reveals the need to modify the design. In the example, extreme data for a guest window include cases where a single guest books rooms for a conference with scores of rooms or where the guest stays for a very long period. Users might need presentation forms with a better overview in such situations.

Reuse old windows or forms. If the old system uses some forms or screens already, should we use them in the new system? It will ease learning the new system, but it might counteract designs with better task support. In the hotel system, for instance, the virtual guest window resembles an invoice, thus helping the novice. However, if we want a good overview for a guest who books scores of rooms, another presentation might be advantageous.

Comparison with traditional approaches. If we had used a traditional data-oriented approach, what kind of physical windows would we have seen? Most likely, the service-charge window would look much the same, because it corresponds closely to the service-type table in the database. However, the guest window would probably become three windows: one with data from the guest record (name, address, and so on), one with the room lines for a specific guest (based on the room state and the room table), and one with the service lines for the guest. (We have seen such solutions in several commercial Enterprise Resource Planning (ERP) products.) The rooms window would probably become a list of room states for a specific date, corresponding to one column of the virtual window. Finally, the breakfast window might not be there at all, because it is not necessary from a data point of view.


Guest Stay#: 714

Name:	John Simpson
Address:	456 Orange Grove
	Victoria 3745
Payment	Visa

	Item		# pers	
7/8	Room 12.	sgl	1	600
8/8	Breakf.	rest	1	40
8/8	Room 11,	dbl	2	800
9/8	Breakf.	room	2	120
9/8	Room11,	dbl	2	800

Breakfast 9/8

R#	In rest	In room
11		2
12	1	
13	1	1

Service charges

Breakf. rest.	40
Breakf. room	60

Rooms					7/8	8/8	9/8	10/8
11	Double	Bath	800	600		O	B	
12	Single	Toil	600		O	O	B	B
13	Double	Toil	600	500		B	B	B

Figure 4. Detailed virtual windows that use graphical design and are filled in with realistic data.

If we use a task-oriented approach, the user first must select a task: book, check-in, and so on. For the book task, the user might be guided first through one window to enter the desired room type and stay period, next through a window to select available rooms in that period, then through a window to enter the guest name and address, and eventually through a confirmation window. The windows for another task (such as room change) might look very different.

Check the design

At this stage the design artifacts are a task list, a set of virtual windows, and possibly a data model. They overlap considerably, which lets us check for completeness and consistency. Here are some useful checks and tests:

Check virtual windows against the data model. Check that all data in the virtual windows exist in the data model or can be derived from it.

CRUD Check. CRUD stands for create, read, update, and delete. Check that it is possible to create, read, and so on all data through some virtual window. Otherwise a window might be missing—and probably a task. (This is a check for oversights—some data might be imported from other systems and thus need no manual update.)

An important purpose of the prototype is usability testing, which can effectively reveal usability problems in the design.

If we check the hotel system, we should notice that service types and their prices cannot be seen in a way where changing them or creating new services makes sense. The solution is to add a new virtual window to show the list of service types and their prices (shown in Figure 4 as Service charges). We also lack at least one task that uses this window. We might call it Maintain Service List, and we should add it to the task list because it is useful for later checking and testing.

Walk through all tasks. Take the tasks one by one and walk through them—manually simulate how to perform each task by means of the virtual windows. As part of this, you might write down the functions needed in each virtual window.

Understandability test with users. An in-depth review of a data model, even with expert users, is rarely possible. They might accept the model, but they rarely understand it fully. With virtual windows, reviews are fruitful because of the detailed graphics and the realistic data.

Show the virtual windows one by one to a user. Ask the user what he or she believes the windows show, whether they show a realistic situation, whether he or she could imagine a more complex situation that would be hard to show on the screen, and whether some data is missing. Perform the review with expert users and ordinary users separately—they reveal different kinds of problems.

For example, in the hotel case, reviews with experts revealed that seasonal prices were missing. Reviews with ordinary users revealed that the mnemonic marking of room state was not obvious. Some users believed that O for occupied meant zero, meaning that the room was free.

You can also ask the user to walk through some tasks. Ask which window he or she would use, what data he or she would enter, what functions he or she would expect to switch to other windows, and so forth. The whole exercise can get quite close to a real usability test,[13,14] although the windows contain no functions (menus, buttons, and so on) at this stage.

Design the dialog

In this step, we design the dialog in detail. This is not part of the virtual-windows approach, but the virtual windows are the basis.[15] Detailed design involves several things:

- Organizing the physical windows. The basic part of this is to adjust virtual windows to physical window size by splitting them, using scroll bars, tabs, and so forth. In some cases, two or more virtual windows might be combined into one physical window or screen.
- Adding temporary dialog data such as search criteria and dialog boxes. The earlier walkthrough and understandability tests are good sources.
- Adding functions to navigate between windows, perform domain-oriented functions, and so on. Again, the walkthrough and understandability tests are good sources. CRUD checks and state-transition diagrams might reveal missing functions.
- Adding error messages.
- Adding help and other guidance.

The design should produce a more or less functional prototype. An important purpose of the prototype is usability testing, which can effectively reveal usability problems in the design.[14] Although early testing of the virtual windows can reveal many problems, new problems creep in during detailed design.

Experience

Over the last six years, we have gathered much experience with the virtual-windows approach in real projects and in courses for designers. Here are some of our observations.

Need for early graphical design

In early versions of the approach, we didn't split virtual-window design into an outline step and a detail step.[15] We observed that some design teams produced excellent user interfaces that scored high during usability tests, while other teams produced bad designs. Furthermore, excellent designs were produced much faster than bad designs. Why?

Gradually we realized that the main difference between good and bad teams was the amount of detail they put into the virtual windows. Both groups could quickly produce the outline version. The bad teams then continued with dialog design, but when they designed the actual user interface, everything collapsed. The outline could not become useful physical windows, fields could not contain what they were supposed to, getting an overview of data was impossible, and so on. The teams had to redesign everything, which resulted in a mess.

The good teams spent more effort designing the graphical details of the virtual windows, filling them with realistic data, and so forth. As part of that, they often modified the outline and grouped data in other ways. These changes were easy to handle at that time, and from then on, things went smoothly. Dialog functions were added, and the physical window design was largely a matter of cutting and pasting parts of the virtual windows.

Adding functions too early

We have observed that designers tend to add buttons to the virtual windows from the very beginning. This goes against the idea of dealing with only data at that stage and delaying functions to later steps. The designers cannot, however, resist the temptation to put that check-in button on the guest window.

We have learned to accept that. It doesn't really harm anything as long as the designers don't focus too much on functions at this stage. Maybe it actually makes the windows more understandable because a button suggests how to use the window. For instance, we have noticed that users better understand that something is a long list if there is a scroll bar.

Forgetting the virtual windows

We have observed many cases where designers made excellent virtual windows, only to forget them when designing the physical windows. The physical design then became driven by available GUI controls and the belief that traditional windows were adequate. The concern for understandability and efficient task support disappeared, and the final user interface became a disaster.

Two things can help overcome this: ensuring that the window designers know the GUI platform to be used (so that they don't propose unrealistic designs) and ensuring that quality assurance includes tracing from the virtual to the final windows.

Examples of projects

Virtual windows stood its first real-life test in 1993 in a project for booking and scheduling classrooms in a university. Classrooms were the most critical resource at that time, and the existing system was entirely manual and an organizational disaster. The university had attempted a computer system, but owing to its heavily data-oriented design, it was not successful.

We designed a new system by means of virtual windows. It became a success and is still the way all room allocation is handled. The database has 20 entity types and 2 million records; there are 150 rooms in 12 buildings and 7,000 courses; and there are two highly complex, 10 moderately complex, and eight simple windows to view.

A recent example is the redesign of a Web-based job match system where applicants could record their qualifications and companies announce jobs. Several such systems exist, but the one in question was the easiest and most widely used. Still, it attracted few users.

In the old design the user had to go through 14 screens to record qualifications. Usability tests showed that most users gave up in the middle of this. The designers used virtual windows to redesign the application, and the result was four screens to record qualifications. Usability tests showed that all users completed the task (except for one, who turned out to be drunk).

Many of our students and clients have successfully picked up the virtual-windows approach.

Many of our students and clients have successfully picked up the virtual-windows approach. It has become the natural way to design user interfaces—for us as well as for them. Over the years they have encouraged us to promote the approach more widely. Until now, however, we have never taken the time to do it, except at a few conferences. We refine the approach a bit every now and then, but basically consider it finished. We have gradually included the approach into more comprehensive methods for user interface design and requirements engineering, but the methods are not yet published in English.

References

1. R. Baskerville, "Semantic Database Prototypes," *J. Information Systems*, vol. 3, no. 2, Apr. 1993, pp. 119–144.

2. C. Janssen, A. Weisbecker, and J. Ziegler, "Generating User Interfaces from Data Models and Dialogue Net Specifications," *Proc. Int'l Conf. Computer–Human Interaction,* ACM Press, New York, 1993, pp. 418–423.
3. S. Lauesen, "Real-Life Object-Oriented Systems," *IEEE Software*, vol. 15, no. 2, Mar./Apr. 1998, pp. 76–83.
4. K.Y. Lim and J.B. Long, *The MUSE Method for Usability Engineering*, Cambridge Univ. Press, Cambridge, UK, 1994.
5. A.G. Sutcliffe, *Human–Computer Interface Design*, Macmillan Press, London, 1995.
6. A. Cockburn, "Structuring Use Cases with Goals," *J. Object Oriented Programming*, Sept./Oct., 1997, pp. 35-40.
7. P. Chen, "The Entity-Relationship Model: Toward a Unified View of Data," *ACM Trans. Database Systems*, vol. 1, no. 1, Mar. 1976, pp. 9–36.
8. E. Yourdon, *Modern Structured Analysis*, Prentice-Hall, New York, 1989.
9. M.B. Harning, "An Approach to Structured Display Design: Coping with Conceptual Complexity," *Proc. 2nd Int'l Workshop Computer-Aided Design of User Interfaces*, Presses Universitaires de Namur, Namur, France, 1996, pp. 121–138.
10. S. Lauesen, M.B. Harning, and C. Gronning, "Screen Design for Task Efficiency and System Understanding," *Proc. Australian Conf. Computer–Human Interaction* (OZCHI 94), CHISIG, Downer, Australia, 1994, pp. 271–276.
11. E. Tufte, *Envisioning Information*, Graphics Press, Cheshire, Conn.,1990.
12. L. Tweedie, "Interactive Visualisation Artifacts: How Can Abstraction Inform Design?" *Proc. Human–Computer Interaction* (HCI 95), Cambridge University Press, 1995, pp. 247–266.
13. J.S. Dumas and J.C. Redish, *A Practical Guide to Usability Testing*, Ablex, Westport, Conn., 1993.
14. A.H. Jorgensen, "Thinking-Aloud in User Interface Design: A Method Promoting Cognitive Ergonomics," *Ergonomics*, vol. 33, no. 4, 1990, pp. 501–507.
15. S. Lauesen and M.B. Harning, "Dialogue Design through Modified Dataflow and Data Modeling," *Proc. Human–Computer Interaction*, Springer-Verlag, New York, 1993, pp. 172–183.

About the Authors

Soren Lauesen is a professor at the IT University of Copenhagen. He has worked in the IT industry for 20 years and at the Copenhagen Business School for 15. His research interests include human–computer interaction, requirements specification, object-oriented design, quality assurance, systems development, marketing and product development, and cooperation between research and industry. He is a member of the Danish Academy of Technical Sciences and the Danish Data Association. Contact him at the IT Univ. of Copenhagen, Glentevej 67, DK-2400 Copenhagen, NV; slauesen@it-c.dk.

Morten Borup Harning is a chief design officer at Open Business Innovation. His research interests include user interface design frameworks, design methods and design notations, user interface design tools, User Interface Management Systems, and human–computer interaction in general. He received his PhD from the Copenhagen Business School. He is a member of IFIP WG 2.7/13.4 on User Interface Engineering and chairs SIGGHI.dk, the Danish Special Interest Group on HCI. Contact him at Dialogical, Inavej 30, DK-3500, Denmark; harning@dialogical.dk.