18

Functional Design

Functional design is the process of translating the needs of our users into a task model that represents the work to be done and then into a functional model that represents the model for our user interface design. Creating the model design is a critical part of user interface development and one that is frequently neglected. In the development of a new interactive application functional design must come first. In this book functional design is placed much later to allow students to begin programming projects early in a semester. However, in actual practice, the functional design process is a critical first step before any code is written.

Several years ago a large software company was loosing market share, though they were still the largest producer in their field. They checked with their customers and found that their products were considered very hard to use. This message went throughout the company and one of their product groups called a consultant to learn how they could improve their user interface. They first presented a document full of screen designs for their new product, which the consultant read and marked up.

They then held a meeting to discuss the findings and plan remedies. The consultant began by asking if anyone in the room had ever used their current product in actual practice. It happened that in addition to the development team one of their support technicians had been invited and raised his hand. When asked about the most common problem calls he received on this product, he replied that there were very powerful controls over access privileges and these privileges had been set wrong. When asked how they were wrong and he explained that it was hard to see the implications that changing a setting might have, therefore people got it wrong and had a hard time figuring out why. The consultant suggested that they change the user interface by adding a display of the impact of access privilege changes so that users could see the implications of some change and respond accordingly. At this point the group manager said “That API was set by the systems team. It is not part of the user interface. We can’t change it.” The discussion moved on to font sizes, field names and other helpful user interface issues. However, the product remained hard to use, the

company continued to lose market share and are now a minor player in their field. Usability is not just about putting a pretty face on the interface.

People buy interactive software because there is something that they want to do. There is some problem for which they want software assistance. The goal of a good user interface is to simplify the user task of solving their problem. Straightening the widget layouts, upgrading the fonts, producing a pleasing color scheme are all important to the user interface but as a colleague of mine recently said "Nicely seasoned slop still tastes bad."

In this chapter we will work through the process of understanding what the user needs and carrying that into a functional design from which our model classes can be implemented. There is a related visual and usability design process, which is the design of the various views of our model. This book, however, is primarily about technology. A most instructive text on functional design is Contextual Design[1]. This chapter will only serve to highlight the issues. To illustrate the functional design process we will use two examples: programming a VCR to record at a later time and buying basketball tickets over the Internet.

Why are we doing this?

Every software project has a set of reasons. Understanding those reasons and keeping them in mind is critical to the success of our user interface. Reasons that seem obvious are not necessarily so. We want anyone to be able to program our VCR to record any show at any time. This sounds like a good goal, but nobody has such a goal. The company building the VCR wants features that sell well in a 15 second demonstration on the showroom floor and in any consumer review article. These two goals seem somewhat tawdry but they are the reality of making money building VCRs. Our original goal statement is also not appropriate for users. Nobody approaches his or her VCR with the goal of recording a particular channel at a particular time. A more realistic user's goal might be "I want to record Star Trek tomorrow while I am at school parent's night". Computer scientists frequently replace intent (record Star Trek) with mechanism (channel and time). The software we are building is mechanism, but to correctly design it we must understand intent.

We may approach development of our basketball ticket system with the stated goal of "helping customers buy the tickets they want as easily as possible." The reality may be that we want to produce the same level of ticket sales as we do currently while spending 60% less on sales personnel. A second goal of our

new system may be to market unused seats by up selling and cross selling customers who are already interested in our sports product. Getting at the real goals rather than the surface goals is important to user interface success.

User interface designs must function in a context of competing forces, needs and constraints. Our project's real goals must reflect those constraints. We do not ignore the altruistic goals of simplicity and ease of use, because they are part of what draws customers to our product. However, the other goals are just as powerful. A major design challenge is to balance these goals, particularly when some forces lend undue prominence to some goals. For example, every large organization has a financial services group. Their stated mission is to provide management with a clear picture of budgets and expenditures. In reality some such organizations are driven by the goals of reducing their own costs (financial services should save the company money) or servicing the needs of auditors (minimize criticism). The results are information systems that enforce static procedures and minimize change. Management, uninformed by auditor-oriented reports frequently hire additional staff to insulate themselves from financial services software (costing money).

> Clarify why we started this project and what are the desired and implied goals. Write them down. Post them where they can clearly be seen. Evaluate design decisions against them.

As much as we would like to create a compelling solution to all of our organization's problems, this is rarely possible. Sometimes such a global rethinking of what we are doing is important. It is much more frequent that we are incrementally improving some aspect of a system. In such situations we need to restate the global goals of the system, but we also need to focus carefully on the problem we are trying to solve and not bury our project in a design morass far beyond our resources.

The steps to creating a good functional design begin with determining the users that the proposed interface should serve. We next work with representatives of these groups to understand their goals (which are not always what they say they are). In particular we look for scenarios of how they accomplish their goals, what information they use and how they decide when they are done. We take this information and develop an object model of the entire system and then choose from that the part that our interface should implement. This leads us to a functional design from which we can build the model of our new interface.

Who will this system serve?

User interfaces are about people interacting with computers. They cannot be effectively designed if we do not know who we are trying to serve. The VCR obviously serves home consumers, but things are not quite that simple. The VCR must also serve the salesperson and the magazine writer. If the salesperson is not served there will be no sale and no consumer use of the system. Fortunately the salesperson's needs and the consumer's needs are very similar. If the "record" feature can occur in the 15 seconds available for a sales demo and can be readily learned by someone who is selling 25 different models of VCR, then it will probably work well for the consumer. However, a consumer may be willing to spend 15-20 minutes of one-time setup of the system where a sales person cannot afford such effort. We need to identify both sets of users and be certain that their needs really are similar. VCR users do not fall in a single uniform lump. There are the elderly with lots of time but poor eyesight, short-term memory and shaky motor skills. There are the techno-teen boys who joy in mastering new gadgets. There are the elderly retired men who see making the thing work as a nice afternoon hobby. Many consumers will want the box to quickly do its job with minimal fuss. We will probably not want to sell our particular VCR to all of these, but rather target a few of them.

The basketball sales application is more problematic in its user base. Obviously there are the people who want to buy tickets. In addition, there is the sales office that must handle problems and coordinate their sales with the new online system. There is the marketing and promotions department who want to put together packages and cross-selling opportunities as part of the ticket sales process. Marketing will also want to track the impact of their promotions and advertising on actual ticket sales. The accounting department will need to track income and process credit card information. The auditors will need a transaction trail to detect abuse. Lastly management will want to know sales numbers. Though the customer is king, all of these users must be served by this system. It lives in an organizational context with which it must interact.

Even when working on a project of limited scope we should still identify all of the users who will be affected by our design and the people that they directly communicate with. We want to go one step broader in our circle of users than the actual people who will interact with the system. The reason for this breadth is to make certain that we have accounted for needs and constraints that may not be obvious to the actual users. Many clerical/data entry people have good knowledge about how they use their current system and its difficulties. However,

they rarely have a good feel for the organizational reasons for why things are done the way they are. Stepping one level beyond them will bring in that context.

> Make a list of all of the types of users that will interface with this system. If there are many types, attempt to group them by similarity of needs. Also note the relative importance of each user type.

Each of the types of users will have different goals when they interact with the new UI design. The VCR sales person has very different goals from the VCR user. The basketball marketing group has different goals from the sports fan. We need to specifically account for each of these diverse sets of goals.

> Refine the project goals to a set of goals for each group of users. Be specific about what each group is trying to accomplish in using the new system.

Various groups of users have varying skills and their own language or terminology. The basketball accounting department will speak in terms of debits, credits, encumbrances and a variety of terms that are foreign to the sports fan. Management personnel sometimes do not possess typing skills or are reluctant to use them on the job because typing indicates loss of status. Event planning people regularly talk about venues while that term has no meaning for many sports fans. When designing a VCR for the elderly user one must remember that their eyesight is poor and they frequently have poor motor skills. Large symbols, numbers and buttons are required in such a market.

Various groups of users frequently have their own symbols and notations for things. Mathematicians, engineers, medical personnel, and musicians all have well developed notations for communicating with each other. There are standard symbols on virtually all VCRs for stop, play, pause, fast-forward and rewind. We need to understand the existence of those symbols and their meaning for various groups of users.

> For each group of users assemble lists of their skills (or lack of skills), terminology and symbology.

What are we trying to do?

Once we know who our users are we know whom to talk with to inform our design. We now need to identify the actual tasks that people are trying to accomplish. These tasks are frequently composed into subtasks.

A user approaches their VCR with the task "Record Star Trek tomorrow". The general form of this task is to watch a particular show on a particular day. This task can be broken down into several subtasks, as shown in figure 18.1.

- Watch show S on day D
  - Find S in a program guide G
    - Identify network (ABC, FOX, NICK)
    - Find start and end time
  - Program VCR
    - Indicate that future recording is required
    - Enter start and end time
    - Translate network into a channel number
    - Enter channel number
  - Load blank tape
    - Put in tape
    - Position tape where recording is to start
    - Determine if there is enough tape to record the show

Figure 18.1 – Task analysis of recording a show on a VCR

It is interesting to compare this task breakdown for a traditional VCR with the new digital equivalents such as TiVo[2]. In a TiVo the program guide is automatically loaded into the device over a network and the videotape has been replaced by a disk drive. The TiVo task structure is as shown in figure 18.2.

- Watch show S on day D
  - Navigate program guide to find S on day D
  - Select record

Figure 18.2 – Task analysis for recording show on TiVo

By paying attention to the user's actual tasks (find and record a show), TiVo makes the process much simpler. The problems of start time, end time and channel are handled automatically. They never were part of the user's goals, only

artifacts of the technology. TiVo eliminates the tape start, end, format and positioning problems by using standard file allocation strategies on a hard disk. No amount of screen, button or display design can compensate for the fundamentally simpler task structure that TiVo offers over the traditional VCR. Seeing the opportunity to increase system usability involves first considering the tasks to be performed.

We can perform a similar task analysis for buying basketball tickets. However, in this task analysis a user may have a variety of tasks as in figure 18.3.

Order tickets for N people for the game on Friday

Order tickets for the next time the Lakers come to town.

Change tickets for this Friday's game to the next Blazers game.

The coach of a kiddie basketball team needs to find a game with a cheap promotion for 12 tickets on a Wednesday or Thursday night before the end of February.

Tickets for next Tuesday have not yet arrived.

Figure 18.3 – Sports fan high-level tasks

From this example we see that "buying tickets" is not as simple as it might first appear. People approach the system with many different tasks and goals in mind. A particular fan could easily perform all of the above tasks at various times. This is only the fan's point of view. We also have all of the other potential users. The goals of creating, pricing, presenting, and evaluating special package promotions carries with it a whole different set of tasks that the marketing group must perform.

> As we talk with users about their tasks, we need answers to four questions:
>
> What is the end goal of the task?
> What are the subtasks?
> What information is required to carry out each task or subtasks?
> How will the user decide if a task has been successfully accomplished?

We need to keep the goal of the task in mind so that we do not get caught up in our current approach to achieving that goal. A process-centric approach to the

VCR recording problem would lead us to the subtask of entering a time. A goal-centric approach tells us that time is only a means to an end. In many recording tasks the user does not care about the time only about a particular show or set of shows.

Breaking down the tasks

Breaking down the tasks into their component tasks is important for a two reasons. First, it helps us to understand what the task actually involves. The high-level fan tasks (figure 18.3) are not very informative in guiding our user interface design. We need more detail. Almost always we will develop these tasks to reflect the approaches with which users are already familiar. This is because users are rarely good at thinking creatively about how they might do something different. We address the "better idea" later when we have captured the information about their current tasks.

In trying to understand a user's tasks many researchers have borrowed the techniques of ethnography from the field of anthropology[3]. The idea is to watch and record how users actually do their current work in their current context. In many ways they are the same techniques used when sitting in the jungle, observing gorillas. There is, however, a fundamental difference between our process and anthropology. An anthropologist is trying to understand the observed behavior while influencing that behavior as little as possible. Interference is a major "no-no" in the science of anthropology. We, however, hope to change the process to make it better. This is a very different mind set.

What information is required?

Modern computing is fundamentally about information more than computation. User interfaces communicate information between people and computers. One of the very important things that we need to know about every task is the information required to complete that task and where that information comes from. Simplifying information flow is probably the most important way to increase the usability of an interactive system. Eliminating the copying of channel/start/end information from the program guide to the VCR is a major contribution of TiVo and similar devices. It simplifies the information flow for the user.

In traditional systems, information flows through an organization on paper. When designing a new user interface the task analysis should collect copies of every type of form, letter or other information. Pay particular attention to notes made in the margins or the use of sticky notes. These are indications of

information not captured by the current system and yet critical to the process. If the current system is already automated, collect screen shots of current interfaces. These are quick ways to identify the required information.

How will the user know they have succeeded in their task?

It is very important to note how a user decides that a particular goal has been met. A huge problem when recording on most VCRs is knowing if the recording is set up correctly. In an effort to save device space, most VCRs do not show the channel/start/end information for the current recording. At most a piece of this information is shown overlaid on the same display devices used for other purposes. A VCR user is rarely confident of their settings. A more subtle problem is knowing how much tape is available and whether the show will fit. Most VCRs provide no indication of this, leaving the user to guess and make very frustrating mistakes.

When a person decides to buy basketball tickets, how do they decide when the task has been completed to their satisfaction? If we think about this problem, people care about which game they bought tickets for, how much they cost and how good are the seats. Items like date, time, teams playing and price are easily displayed. How do people evaluate the quality of their seats? Most people would rather be closer than farther away. Most of us balance good seats with price. Most of us prefer chair seats to bench seats. What a lot of us care about is the view of the court from our seat. Displaying a photo of the court from seat location would greatly simplify seat choice. This approach would directly address the user's goal of seat quality. Understanding how the user evaluates success tells us a lot about the kinds of information that we should present and how we should present it.

> Talk to members of all of the user types and make lists of tasks to be performed along with: who performs the task, what information they need to perform the task, and how will they decide that the task has been performed successfully.

Scenarios of use

While developing a task model (the set of tasks that users desire to accomplish), it is very helpful to create and save scenarios of how users will accomplish their tasks. These scenarios should involve a specific user with a specific goal. We then go through their task step by step. Many such scenarios can be developed from ethnographic studies of current systems.

These scenarios serve four purposes. First, they force us to be very specific about how the work actually gets done. Secondly, by working through the scenario we can carefully identify every function to be performed and every piece of information required to perform that function. Third, these scenarios make the design process real for potential users. Users do not work well with abstract descriptions. Abstraction is a computer science skill that comes with training. Potential users need concrete examples to critique and improve. Lastly we will use our scenarios to evaluate our functional designs to see if they actually accomplish all that we have planned.

While constructing our scenarios we need to note the exceptions that can occur. At each step, what are the things that might go wrong? These include missing information, incorrect entry, interruptions, or misunderstandings. Quality systems support all of the work, not just the simple clean portions. Good user interfaces help prevent or highlight errors early rather than later. Understanding these errors is very important.

> For each task prepare several scenarios of a user accomplishing the task. Note the functions and information required as well as the exceptions that might arise.

Object-based Task/Function Models

User interfaces are primarily designed around the manipulation of information. Even where extensive calculation is involved, the user interface consists modifying the control information for the calculation, initiating the calculation and then reviewing or monitoring the output. Recognizing the info-centric nature of user interfaces we design models based on the task's information objects.

There are a variety of representation schemes that can be used in model design, such as Entity-Relationship Diagrams[4] or the Unified Modeling Language (UML)[5]. For smaller projects an abbreviated version of the class definitions of your favorite object-oriented language will serve. What is needed is to clearly capture the types of information objects required, their attributes and their methods. We do not want to implement them yet; we just want to understand them.

For our VCR example the types of information are program guides, days, lists of shows, shows, times and VCRs. Figure 18.4 shows a model in C#-like syntax.

```
class ProgramGuide
{     ShowList find(String nameOfShow);
      ShowList find(String nameOfShow, Day whichDay)
}
enum DayOfWeek{Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday}
enum Month(January, February, … December}
class Day
{     Day(DayOfWeek dayThisWeek);
      Day(Month month, int day);
      Day Today();
      Day Tomorrow();
}
class ShowList
{     int numberOfShows(); // how many shows in the list
      Show this[int whichShow]; //index into the show list
      ShowList filter(Day whichDay); // filters a show list to those on a particular day
      ShowList filter(string showName): // filters a show list to a particular show name
}
class Show
{
      string network; //which network is it on
      int channel; //which channel is it on
      string showName; // what is the name of the show
      Time start, end;
      Day dayOfShow;
}
class Time
{     int hour,  minute;
}
class VCR
{     void record(Show whichShow);
      int channel;
      Day recordDay;
      Time start, end;
}
```

Figure 18.4 – Model for VCR programming

In our model design, we have not accounted for many of the issues required for a full implementation. There are no data structures. There are none of the notifications required to make a model work in MVC. We have used fields on the classes without paying attention to accessor methods to control them. What we want is a clean specification of the function of our desired interface.

The model design for our basketball ticket system will be much more complex. There are venue calendars, events, classes of seating, promotional packages, group discount packages, advertising material (the HTML promoting each event), transactions, seat availability maps, customers/credit cards. Great care must be taken in developing this model because it sharply impacts what will be possible once it is installed. A simplistic view of ticket sales would ignore

promotional packages. However, such packages are very important in filling less important games or those high priced lower bowl seats that are behind the basket. Selling tickets over the web includes presenting advertising of the games. If we are very simplistic in how we address the problem of advertising on our web site, we my lose our anticipated cost reductions by increased advertising costs. Capturing all of these interactions before casting our design into code is very, very important.

> Construct an object model by creating classes for each of the types of information extracted from the scenarios. For each class specify the constructors, fields, and methods required to perform the processes identified in the task analysis. Walk through all scenarios with your object model to make certain it is complete.

Assigning agency

To this point we have cast a rather broad net in looking at the tasks that our new user interface must perform. We have looked at neighboring tasks, related users and the rest of our interface's context of use. However, at some point we must decide what will actually be built. It is rarely practical to automate our entire model. We must pick the key parts of the model and provide those in our system. It is also frequently the case that many parts of the system already exist. The assignment of agency is to determine who or what is responsible for each class of information object in our model.

In our VCR example, the traditional VCR design relegates the ProgramGuide,ShowList and Show to a paper document. The VCR implements only the Day(DayOfWeek) and the VCR classes. Operations like Today() and Tomorrow() are done by the user remembering what day of the week it is and making the right choice. The user is responsible for translating the show information into the required date and time.

A prime reason for casting a wide net during task analysis is the discipline it imposes when assigning agency. Whenever we designate some task step as outside of our system, we must account for how information will cross the boundaries into and out of our system. These information flows across boundaries are extremely important and are a major source of system errors. The information flows between user and system define the functionality of the user interface.

This assignment of agency and communication of information across system boundaries is particularly problematic in business settings. The traditional

approach is to use paper to cross boundaries. The process of my buying a new laptop will illustrate this. In the traditional approach the follow steps would occur.

- I decide what kind of laptop I want, write it down and give it to our department's support staff.
- The support staff types up a purchase order, gets it signed by the chair and dean and sends it to purchasing.
- Purchasing contacts the vendor and asks for a quote.
- Vendor prints and mails a quote.
- Purchasing office types the quote information into their ordering system.
- Ordering system prints an order that is mailed to the vendor.
- Vendor types the order into their sales system.
- Laptop is built and shipped.
- Invoice is printed and mailed to our purchasing

The process actually goes on endlessly through shipping, receiving and our own computing staff. At each stage, paper is used to communicate and information is typed, retyped and retyped again. Through all of these steps there is confusion about what is ordered and when it will be delivered. There are many systems with many user interfaces and many phone calls to coordinate and track problems in the process.

Our current system is that I get on the vendor's web site and select the configuration of laptop that I want. The system generates a quote number, which I email to our support staff. They get signed approvals and send the quote number to purchasing who forwards it directly to the vendor. The vendor builds according to the system that I specified and ships directly to me. At any time I can check the quote to see the status of when I will receive the system. Invoices and payments happen automatically and digitally. This system is much more usable because information flows have been optimized to meet my task (buying what I want) rather than the cross organization communication task. Thinking through the functional design of the process rather than fiddling with screen layouts was what provided the great leap in usability.

Assign agency to all parts of the object model. Who or what is responsible for performing all of the operations and handling all of the information in the model.

Carefully review information flows between the new implementations and other systems and people.

Functional Design

Having assigned agency to the classes, fields and methods of our task-based model we can now produce our functional design. If we take the traditional VCR approach the function design is as shown in figure 18.5. Note that this functional design ignores all of the other functions of a VCR. To do a real design we need to account for these others.

```
enum DayOfWeek{Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday}
class Time
{     int hour,  minute; }
class VCR
{
      int channel;
      DayOfWeek recordDay;
      Time start, end;
}
```

Figure 18.5 – Traditional functional design for VCR recording

Considering all of your task analysis, object-modeling and agency construct a class, constructor, field, method model for exactly what your user interface must do.

View sketching to evaluate functional design

Having a functional design like that shown in figure 18.5 lays the foundation for our user interface implementation. We now know what we are trying to build. However, our users do not know what we are trying to build. Presenting prospective users with a functional design like that shown in figure 18.5 will not produce interesting results. They will have very little idea how this notation relates to their problem or its solution. This functional model needs to be translated into view sketches. View designs are simply pictures of how the information might appear to the user. Designing the views is not properly part of functional design. However, users cannot evaluate designs any other way. They must see it in concrete terms.

When sketching prospective views of our functional design it is important that the process be as lightweight as possible. We need to be able to generate and evaluate many designs quite rapidly. If our application can be built out of standard widgets we can use an interface design tool to rapidly layout widgets to hold the information and invoke the methods specified by our functional design. This can be prepared in minutes, printed and shown to prospective users. Using these printed mock-ups, we can work through our scenarios with actual users to see if they can accomplish all of the tasks in a convenient fashion or if some information/capability is missing.

We also need to remember that we are evaluating functional design not presentation design. The key question at this point is can we conveniently do what we need to do. A pitfall in view sketching is that people start to get caught up in issues of color, font selection, layout etc. This is the wrong time for such discussions. What we are trying to do is project the functionality of the system in a form the users can understand and evaluate. Having said that, we must not be purists. We should accept input about screen design, terminology and layout because we will eventually need such information. The key is to not lose sight of the larger goal, which is get the functionality right before spending effort on smaller details. This is the time for considering diverse alternatives for modeling the problem.

A second pitfall in view sketching is the presumption of "doneness" and the seduction of increased fidelity. If we are using an interface design tool to present our functional design, the design starts to look very finished. If our users are a little naïve about software development, the finished look of the sketch implies that the product as a whole is almost done. The finished look of the sketch also starts to cause a mind set for the way the functionality "must" be implemented. Early rigidity cuts off potentially better alternatives. Some design tools provide a "sketchy" look that purposely makes the design look rough and unfinished[6]. The Silk system allows designs to be drawn a active sketches[7]. It is also tempting when sketching with an interface design tool to keep improving the fidelity of the interface making it more and more real. This produces programmer rigidity when time investment in a particular design starts to block consideration of alternatives.

One approach to prevent such early commitments is "paper prototyping" [8]. In this approach, software is avoided and paper sketches of view designs are used exclusively. Using paper rather than the computer, there is no implication of "doneness". It is also easy to write on paper designs and thus change them on the fly during the design process. This has the additional advantage of being

accessible to designers with limited programming skills but good understanding of users and their problems. These are all techniques for getting user feedback on whether our functional designs actually capture the problem we are trying to solve.

> Render the functional design into a series of "view sketches" that show the functional capabilities as a user interface. Use these sketches to evaluate scenarios and do design walkthroughs with prospective users.

Translating to a Model Implementation Interface

Having created a functional design model and a series of tentative view sketches, we can translate these into an actual model implementation interface. What we are implementing is not the functionality of the model but rather the methods that it will use to expose the functionality to view/controllers and to other parts of the system. This design of the interface methods for the functionality allows view/controller development to proceed independently from application functionality development.

The functional model classes translate directly into implementation model classes. Fields must be translated into accessor functions. Methods usually translate directly. In many cases we will need to flesh out our data structures. While doing design we wave our hands over "lists" and other structures. At this point we need to design how our views will actually access information they will need to present the model. Our view sketches are important in this process because they, along with our functional design, will tell us about the information needs of the views.

Considering our view sketches we can also design our notification strategy. If our views are simple we can go with a single "model changed" method. If they are more complex we may need a more specific "this element changed" notification. We do not need detailed view designs to make these decisions, but a general idea of how the information will be viewed can help a lot in designing a notification strategy. Now is also the time to make certain that the notification strategy is robust and easily used with the code and concentrated in a few places.

> Translate functional design into model API by defining all of the classes and methods of the model. This includes accessor methods for data and the view notification strategy.

A note about testing

Building a robust testing facility for user interfaces is quit difficult. Regression testing involves the creation of self-checking tests that can be automatically run whenever changes are made. This is extremely important to large development projects. The problem when testing interactive programs is the creation of an “artificial user”. How do we test for users having input mouse points at particular locations when layouts allow widgets to shift and move? How do we test for “the display shows the right stuff”? One approach that helps is to carefully develop a model that contains most of the functionality. With such a model and a clean model interface, a series of regression tests can be developed that evaluate the changes, the notifications and a variety of other properties of the model without incurring the difficulties of simulating a user. Trying to test everything interactively can be frustrating, error prone and not very complete.

A Historical Prospective

We have devoted an entire chapter to the problem of designing exactly what our user interface should do rather than how it should do it. There is a great temptation to either skip this step or simply replicate an older design using new technology. A piece of history is enlightening here.

When Thomas Edison was inventing the electrical power industry, manufacturing was primarily powered by steam. Factories were constructed by building a large steam engine at one end of a factory and having it power a shaft that ran the length of the factory. Each machine tool was connected to this shaft by means of a belt that would transfer power from the shaft to the tool. The placement of these tools in the factory was dictated by their proximity to the power shaft. When electricity replaced steam most factories replace their steam engines with very large electric motors to turn the shaft. Otherwise the structure of the factory remained the same.

One of the paradoxes was that for the first 20 years of electrical power, there was no increase in productivity in the economy. Gradually smaller electric motors were developed. They soon became small enough to put an individual motor on each power tool. Power was then transmitted through the factory by wires that could be run anywhere. Factories could be restructured around the workflow rather than the power shaft. The nature of how factories and their work were structured changed and became more effective. Finally the productivity of the economy took a sharp increase. As long as electrical power was used without restructuring the work, there was little increase in productivity.

The same productivity paradox has been reported for computing[9]. For the first 20 years of computing the economy did not show an increase in productivity. Personal computers let secretaries produce nicer looking documents and edit them more quickly. However, until most of the white-collar workforce learned to type and found that typing and revising their own letters was faster than multiple drafts through a secretary, the number of secretaries required per person remained unchanged and there was no net gain. The work change to reduce staffing was required before benefits were realized. Until businesses like FedEx, Dell and WalMart changed the way their industry worked to take advantage of the new opportunities provided by computers, the economy did not shift.

As we develop user interfaces, it is most important to remember that the greatest increase in "usability" will come when the work itself becomes simpler, not just beautifying the screens. Beginning with a careful look at the fundamental work being done and then modeling that work in easy to understand user interfaces is key to effective design of interactive systems.

---

[1] Beyer, H. and Holtzblatt, K., *Contextual Design: A Customer-Centered Approach to Systems Designs*, Morgan Kaufmann (1997).

[2] http://www.tivo.com/

[3] Crabtree, A., *Designing Collaborative Systems: A Practical Guide to Ethnography*, Springer (2003).

[4] Thalheim, B., *Entity-Relationship Modeling: Foundations of Database Technology*, Springer (2000).

[5] Larman, C., *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*, Prentice Hall (2004).

[6] Hudson, S. E. and Tanaka, K. "Providing Visually Rich Resizable Images for User Interface Components", *User interface Software and Technology (UIST '00)*, ACM (2000), pp 233.

[7] Landay, J. A. 1996. SILK: Sketching Interfaces Like Krazy. *Human Factors in Computing Systems(CHI '96)*. ACM (1996), 398-399.

[8] Snyder, C. *Paper Prototyping: The Fast and Easy Way to Design and Refine User Interfaces*, Morgan Kaufmann (2003).

[9] Landauer, T. K., *The Trouble with Computers: Usefulness, Usability and Productivity*, MIT Press (1995).