Course Code and Name	**COM 216 SOFTWARE AND INTERFACE DESIGN**	
Credit Units	3	
Pre-requisites		
Purpose of the Course	To introduces students to the principles and methods for the design of software systems in professional contexts.	
Expected Learning Outcomes	At the end of this course, the student should be able to • undertake system design in a methodical manner • proceed from a general system or product requirement to a design that addresses user needs • develop design models and prototypes in an iterative manner recognizing managerial risks • evaluate interactive systems, including identification and correction of faults.	
Course Content	Mental models, leading to gulfs of execution and evaluation, Observing and describing the needs of users in context, Methods for iterative modelling and prototyping, Observational and experimental methods for usability evaluation.	
Mode of Delivery	Lectures, discussions, laboratory demonstration and tutorials.	
Instructional Material and/or Equipment	Textbooks, White board, Chalkboard, Handouts, Computers.	
Course Assessment	**Type**	**Weighting (%)**
	Examinations	70%
	CATS	30%
	Total	100%
Recommended Reference Material	• Interaction Design: Beyond human-computer interaction by Helen Sharp, Yvonne Rogers & Jenny Preece • Software Engineering by Pressman (multiple editions)	

COM 216: SOFTWARE & INTERFACE DESIGN LECTURE NOTES

MENTAL MODELS

What are mental models?

Mental models are psychological representations of real, hypothetical, or imaginary situations. They were first postulated by the American philosopher Charles Sanders Peirce, who postulated (1896) that reasoning is a process by which a human

"examines the state of things asserted in the premisses, forms a diagram of that state of things, perceives in the parts of the diagram relations not explicitly mentioned in the premisses, satisfies itself by mental experiments upon the diagram that these relations would always subsist, or at least would do so in a certain proportion of cases, and concludes their necessary, or probable, truth."

The Scottish psychologist Kenneth Craik (1943) proposed a similar idea; he believed that the mind constructs "small-scale models" of reality that it uses to anticipate events, to reason, and to underlie explanation. Like pictures in Wittgenstein's (1922) "picture" theory of the meaning of language, mental models have a structure that corresponds to the structure of what they represent. They are accordingly akin to architects' models of buildings, to molecular biologists' models of complex molecules, and to physicists' diagrams of particle interactions.

Since Craik's insight, cognitive scientists have argued that the mind constructs mental models as a result of perception, imagination and knowledge, and the comprehension of discourse. They study how children develop such models, how to design artifacts and computer systems for which it is easy to acquire a model, how a model of one domain may serve as analogy for another domain, and how models engender thoughts, inferences, and feelings.

HOW WE REASON

The theory of mental models rests on simple principles, and it extends in a natural way to inferring probabilities, to decision making, and to recursive reasoning about other people's reasoning. We can summarize the theory in terms of its principal predictions, which have all been corroborated experimentally. According to the model theory, everyday reasoning depends on the simulation of events in mental models (e.g., Johnson-Laird, 2006). The principal assumptions of the theory are:

1. **Each model represents a possibility.** Its structure corresponds to the structure of the world, but it has symbols for negation, probability, believability, and so on. Models that are kinematic or dynamic unfold in time to represent sequences of events.
2. **Models are iconic insofar as possible**, that is, their parts and relations correspond to those of the situations that they represent. They underlie visual images, but they also represent abstractions, and so they can represent the extensions of all sorts of relations. They can also be supplemented by symbolic elements to represent, for example, negation.
3. **Models explain deduction, induction, and explanation.** In a valid deduction, the conclusion holds for all models of the premises. In an induction, knowledge eliminates models of possibilities, and so the conclusion goes beyond the information given. In an abduction, knowledge introduces new concepts in order to yield an explanation.
4. **The theory gives a 'dual process' account of reasoning.** System 1 constructs initial models of premises and is restricted in computational power, i.e., it cannot carry out recursive inferences. System 2 can follow up the consequences of consequences

recursively, and therefore search for counterexamples, where a counterexample is a model of the premises in which the conclusion does not hold.

5. **The greater the number of alternative models needed, the harder it is:** we take longer and are more likely to err, especially by overlooking a possibility. In the simulation of a sequence of events, the later in the sequence that a critical event occurs, the longer it will take us to make the inference about it.
6. **The principle of truth: mental models represent only what is true**, and accordingly they predict the occurrence of systematic and compelling fallacies if inferences depend on what is false. An analogous principle applies to the representation of what is possible rather than impossible, to what is permissible rather than impermissible, and to other similar contrasts.
7. **The meanings of terms such as 'if' can be modulated by content and knowledge**. For example, our geographical knowledge modulates the disjunction: Jay is in Stockholm or he is in Sweden. Unlike most disjunctions, this one yields a definite conclusion: Jay is in Sweden.

The theory accounts for the informality of arguments in science and daily life, whereas logic is notoriously of little help in analyzing them. If people base such arguments on mental models, then there is no reason to suppose that they will lay them out like the steps of a formal proof. The theory of mental models, however, is not a paragon. It is radically incomplete; and it is likely to have problems and deficiencies. Proponents of rule theories have accused it of every conceivable shortcoming from blatant falsehood to untestability. It postulates that human reasoners can in principle see the force of counterexamples, and indeed people are able to construct them — a competence that is beyond the power of formal rule theories to explain. The model theory may well be overturned by counterexamples predicted by a superior theory. In which case, it will at least have had the virtue of accounting for its own demise

Mental Models and Learnability:

Many items we use every day would appear meaningless without training. Consider the controls on a cooker: most of the controlled items are distant from the associated switches and knobs, and the operation of switches vary - some are rotary, some push and others slide. Often the layout of the controls does not match the layout of the hobs, grill and oven Some electric ovens have the grill as part of the oven, and have more than one knob to control the associated heating elements. Which knob controls which item, and how do they interrelate? It is only by using such a system and creating a mental model, and perhaps using skills learned from other such systems that a person can learn how to operate it. One important aspect of a child's upbringing is the familiarization with systems which are quite complex, but which when learned, eventually form part of everyday life.

Complexity

In terms of computer software, more control by the user is desirable, but an increase in the number and complexity of tools for increasing the user's control has a negative utility [Fischer & Lemke, 1988], and this is true in other contexts as well. Where a large number of controls are required, it may be better to reduce the apparent complexity of a system by careful positioning of the controls - hiding those that are not often required for example. Quite often a small fraction of the functionality of complex systems is used. It can be seen that some complex systems are easy to operate - consider driving a car for example. There are a large number of components to a car, and yet the user of the car needs to be directly aware of only a subset of them, the rest only being

important occasionally. Compare this with the modern digital telephone found in many offices (figure 3.2): many fewer controls, and yet despite training sessions, many people cannot make their telephone perform more than the basic functions.

fig. 3.2 Digital office-telephone.

When a person uses a control to perform an operation they produce a mental model of the *mapping* between the control and the result. This mapping need not be the actual one, as long as it allows the person to remember that their action leads to the result. There can be problems, however, if the mapping that is produced leads to some incorrect assumptions about the underlying system. For example, in many cars the brake light on the dashboard serves as an indicator that the hand-brake is on - courtesy of a switch on the brake, and also to warn of problems with the braking system. Most drivers never see the light used as a warning, and so if it is lit could be forgiven for thinking that the switch on the hand-brake was not working properly. This conclusion could be a dangerous one. A designer of a control needs to be aware that users will need to create the mapping, and help them by perhaps employing natural mappings - physical analogies and cultural standards [Norman, 1988]. For example, suppose that the designer wants a control which allows the user to adjust an amount of something. A natural mapping for this is the cultural standard that a rising level means more. There are two mental models associated with every system (see figure 3.3). The first model is the *design model*. This is the model that the designers have of how the system should work. The design model is interpreted by the programmer to produce a *system image*. The user sees the system image and creates a *user model* of it in order to be able to work with it. This user model may sometimes be very different from the original design model, and may carry with it ideas from other designs that the designer never intended to be part of this one.

This can lead the user to actions which are inappropriate when examined in light of the design model, but which are perfectly reasonable in the user model.

fig. 3.3 the two mental models associated with a system.

Any mismatches between the user's mental model, and the designer's mental model can be reduced by the designer:

Develop a simple, smooth design model, reflective of the needs of the user, not the limitations of the hardware or the difficulties of the coding process. [Tognazzini, 1991].

The Gulfs of Execution and Evaluation

The user's goals and intentions are separated from concepts and operations represented in the system by two gulfs: the gulf of execution and the gulf of evaluation (see figure 3.4).

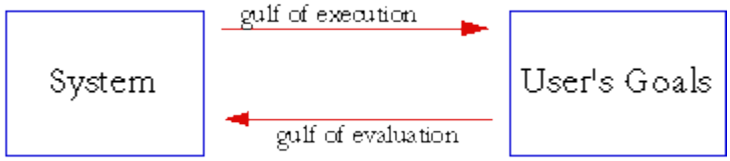


fig. 3.4 The gulfs of execution and evaluation between the user and the system.

The gulf of execution refers to the conversion of the user's goals and intentions into actions that are required by the system. The gulf of evaluation signifies the problem of representing the system's concepts and operations in a form which can be interpreted by the user. According to Ziegler and Fähnrich [Ziegler & Fähnrich, 1988], interfaces which rely on the direct manipulation of objects help bridge these gulfs by allowing the user to utilise elements of the system's output as components of the user's input language. In a direct manipulation system such as the desktop environment of the Macintosh, the user can select an output from the system and continue to manipulate it directly, thereby reducing the gulfs of execution and evaluation.

Feedback

Feedback is an important concept in the design of controls. The user needs to be aware that an action has been performed. Consider how much harder it must be to talk to someone if you cannot hear your own voice. The feedback given must also be appropriate to the task. Some automated teller machines at banks give only audio feedback as you enter your PIN on the keypad. Such systems are difficult to use if you cannot hear the beeps due to high levels of background noise from cars etc. As computers have become more powerful, the sort of feedback that they are able to give has become more varied. Consider the action of copying a file from one directory to another. In early, less powerful computer systems there would be little feedback; perhaps a message saying the file had been successfully copied once the copying was complete. If you wished to stop the copying once it had started there was no easy way to determine how much of the file had been copied, and how much was left to do. In modern graphical computer systems, the copying process may appear as a window indicating the progress of the copy (figure 3.5), and perhaps an animation of the icon for the file moving from one place to another. This gives immediate feedback to the user that an action is taking place, even if it takes an appreciable amount of time. Some systems even present a button to enable the copying to be stopped - the user is always aware how much is left to do, and can see that it is perfectly feasible to stop the copying.

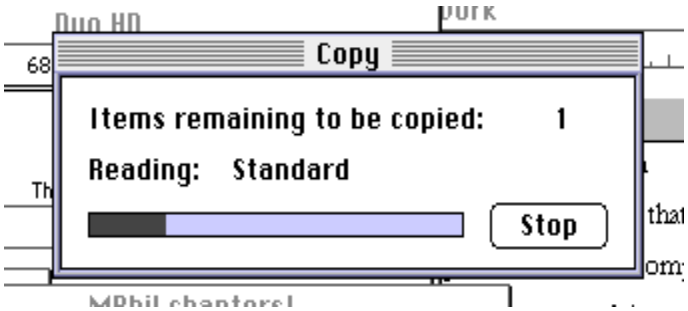


fig. 3.5 - the Finder's standard file-copy progress dialog box.

Feedback is important even if the amount of time taken to do something is increased by providing the feedback. For example, some CPU-intensive applications on the Macintosh draw a progress dialog-box (such as that in figure 3.5 above) while they perform their calculations. There is an amount of CPU power used to repeatedly draw the progress-bar, and this is obviously not available to do the work whose state of completion the progress-bar is indicating. It would seem at first sight that the best thing to do is to utilise all the CPU power for doing the calculation, but by displaying a progress-bar, the user has feedback about how much of the operation is left to perform, and by retaining the user's interest the calculation may actually seem quicker as a result.

Summary:

Mental models are one of the most important concepts in human–computer interaction (HCI). Indeed, we spend a good deal of time covering their design implications in our full-day training course on User Interface Principles.

Here, I'll report a few examples from our usability studies. Not coincidentally, using concrete examples often helps people understand abstract concepts (such as "mental models").

First, though, you have to suffer one bit of theory — namely the **definition of mental models**. A mental model is **what the user believes** about the system at hand.

Note the two important elements of this definition:

- A mental model is based on **belief, not facts**: that is, it's a model of what users know (or think they know) about a system such as your website. Hopefully, users' thinking is closely related to reality because they **base their predictions** about the system on their mental models and thus plan their future **actions** based on how that model predicts the appropriate course. It's a prime goal for designers to make the user interface communicate the system's basic nature well enough that users form reasonably accurate (and thus useful) mental models.
- Individual users each have their own mental model. A mental model is internal to each user's brain, and different users might construct different mental models of the same user interface. Further, one of usability's big dilemmas is the common **gap between designers' and users' mental models**. Because designers know too much, they form wonderful mental models of their own creations, leading them to believe that each feature is easy to understand. Users' mental models of the UI are likely to be somewhat more deficient, making it more likely for people to make mistakes and find the design much more difficult to use.

Finally, mental models are **in flux** exactly because they're embedded in a brain rather than fixed in an external medium. **Additional experience** with the system can obviously change the model, but users might also update their mental models based on **stimuli from elsewhere**, such as talking to **other users** or even applying **lessons from other systems**.

Remember Jakob's Law of the Internet User Experience: Users spend most of their time on websites *other*than yours. Thus a big part of customers' mental models of your site will be influenced by information gleaned from other sites. People expect websites to act alike.

Mixed-Up Mental Models

Many of the usability problems we observe in studies stem from users having mixed-up mental models that**confuse different parts of the system**.

For example, the word "Google" is usually the top query at other search engines, and words like "Yahoo" and "Bing" score high on Google. Why, oh why, do people *search* for a website if they already know its name? Why not just type, say, www.bing.com into the URL field?

The reason is that many users have never formed an accurate model of how the "type-in boxes" on their screen function. When they type stuff into a box, they sometimes get where they want to go. What to type where and exactly how each type-in box functions, however, are often beyond their ken.

The inability to distinguish between similar type-in boxes is a key reason for the guideline to avoid multiple search features. When a website or intranet has several search engines on the same page, users often don't know the difference. They'll enter their query into whatever box catches their fancy and assume that the site doesn't have the answer if nothing comes back. (In reality, they might have used a specialized search that didn't cover everything.)

The one exception to this guideline is that it's preferable to have a separate search on intranets for the employee directory, because in most users' mental models, finding a colleague's contact details doesn't constitute "search." It's looking up a *person*, not searching for *information*. In contrast, of course, any developer mentally models the employee directory as one more database and a phone number as a piece of data. As I said: users and the design team have very different mental models, and you have to understand the users' model to design something that works in the real world.

Users don't just confuse search fields; many less-techy users **don't understand the differences between**many other common features:

- Operating-system windows vs. browser windows
- A window vs. an application
- Icons vs. applications
- Browser commands vs. native commands in a web-based app
- Local vs. remote info
- Different passwords and log-in options (users often log in to other websites as if they were logging in to their email)

Mental Model Inertia

Netflix is a mail-order service for renting movies on DVD. However, Netflix works differently than typical e-commerce sites, which caused problems when we tested it with new users in our project about famous sites' usability:

- When users added a film to their Netflix "queue," they used a mental model of an e-commerce shopping cart to **predict what would happen: nothing**. Adding stuff to the cart doesn't cause you to receive that item in the mail. You first have to proceed through checkout and confirm that you want it.
- In reality, however, Netflix will **immediately mail** you the DVD that's on top of the queue. Later, when you mail it back, they'll send you the next movie in your queue, without you having to go to the site and do anything. That's why they have the "queue" feature instead of a standard shopping cart.

There's great **inertia in users' mental models**: stuff that people know well tends to stick, even when it's not helpful. This alone is surely an argument for being conservative and not coming up with new interaction styles.

On the other hand, sometimes you do need to innovate, but it's best to do so only in cases where the new approach is clearly vastly superior to the old, well-known ways. Netflix is obviously a successful company, and its innovation of sending customers a steady stream of movies from a queue was a major reason for this success.

When you do something new on the web, you face an immense design challenge: How do you explain the new concept such that users have a living chance of constructing a valid mental model of the site?

It's amazing how one misconception can thwart users throughout an entire session and cause them to systematically misinterpret everything that happens on the site. Through failure after failure, they never question their basic assumptions. This is yet another argument for complying with preexisting user expectations whenever possible. If you don't, then make certain that you're

clearly explaining what you're doing — while also realizing that you face the added challenge of users' reluctance to read very much.

Acting on Mental Models

Understanding the concept of mental models can help you **make sense of usability problems** in your design. When you see people make mistakes on your site, the reason is often because they've formed an erroneous mental model. Although you might be unable to change the UI at that point, you can teach users a more accurate mental model at an earlier stage of the user experience. Or, you might have to acknowledge that users won't understand certain distinctions and then stop making those distinctions.

In case of a mental-model mismatch, you basically have two different options:

- **Make the system conform** to users' mental models — assuming most models are similar. This is the approach we usually recommend to fix IA problems: If people look for something in the wrong place, then move it to the place where they look for it. Card sorting is a useful way to discover users' mental model of an information space so that you can design your navigation accordingly.
- **Improve users' mental models** so that they more accurately reflect your system. You can do this by, for example, explaining things better and making labels clearer to make the UI more transparent (even though the underlying system remains unchanged).

Mental models are a key concept in the development of instructions, documentation, tutorials, demos, and other forms of user assistance. All such information must be short, while teaching the key concepts that people need to know to make sense of the overall site. It's sometimes worth trying a short **comic strip**; research has shown that mental-model formation is enhanced when concepts are simultaneously presented in both visual and verbal form.

One of the main reasons I like the **thinking aloud method** of user testing is that it gives us insights into a user's mental model. When users verbalize what they think, believe, and predict while they use your design, you can piece together much of their mental model. There are also more advanced knowledge-elicitation methods for gaining deeper insights into mental models, but fo+r most design teams, a few quick think-aloud sessions will suffice. In any case, simple user testing is certainly the first step to take if you suspect that erroneous mental models are costing you business.

Introduction to Human Computer Interaction

Introduction

HCI (human-computer interaction) is the study of how people interact with computers and to what extent computers are or are not developed for successful interaction with human beings.
As its name implies, HCI consists of three parts: the user, the computer itself, and the ways they work together.

User

By "user", we may mean an individual user, a group of users working together. An appreciation of the way people's sensory systems (sight, hearing, touch) relay information is vital. Also, different users form different conceptions or mental models about their interactions and have different ways of learning and keeping knowledge and. In addition, cultural and national differences play a part.

Computer

When we talk about the computer, we're referring to any technology ranging from desktop computers, to large scale computer systems. For example, if we were discussing the design of a Website, then the Website itself would be referred to as "the computer". Devices such as mobile phones or VCRs can also be considered to be “computers”.

Interaction

There are obvious differences between humans and machines. In spite of these, HCI attempts to ensure that they both get on with each other and interact successfully. In order to achieve a usable system, you need to apply what you know about humans and computers, and consult with likely users throughout the design process. In real systems, the schedule and the budget are important, and it is vital to find a balance between what would be ideal for the users and what is feasible in reality.

Human Computer Interface (HCI) was previously known as the man-machine studies or man-machine interaction. It deals with the design, execution and assessment of computer systems and related phenomenon that are for human use.

HCI can be used in all disciplines wherever there is a possibility of computer installation. Some of the areas where HCI can be implemented with distinctive importance are mentioned below −

- **Computer Science** − For application design and engineering.
- **Psychology** − For application of theories and analytical purpose.
- **Sociology** − For interaction between technology and organization.
- **Industrial Design** − For interactive products like mobile phones, microwave oven, etc.

The world’s leading organization in HCI is ACM − SIGCHI, which stands for *Association for Computer Machinery − Special Interest Group on Computer–Human Interaction*. SIGCHI defines Computer Science to be the core discipline of HCI. In India, it emerged as an interaction proposal, mostly based in the field of Design.

The Goals of HCI

The goals of HCI are to produce usable and safe systems, as well as functional systems. In order o produce computer systems with good usability, developers must attempt to:

- understand the factors that determine how people use technology
- develop tools and techniques to enable building suitable systems
- achieve efficient, effective, and safe interaction
- put people first

Underlying the whole theme of HCI is the belief that people using a computer system should come first. Their needs, capabilities and preferences for conducting various tasks should direct developers in the way that they design systems. People should not have to change the way that they use a system in order to fit in with it. Instead, the system should be designed to match their requirements.

Usability

Usability is one of the key concepts in HCI. It is concerned with making systems easy to learn and use. A usable system is:

- easy to learn
- easy to remember how to use
- effective to use
- efficient to use
- safe to use
- enjoyable to use

Shneiderman's Eight Golden Rules

Ben Shneiderman, an American computer scientist consolidated some implicit facts about designing and came up with the following eight general guidelines −

- Strive for Consistency.
- Cater to Universal Usability.
- Offer Informative feedback.
- Design Dialogs to yield closure.
- Prevent Errors.
- Permit easy reversal of actions.
- Support internal locus of control.
- Reduce short term memory load.

These guidelines are beneficial for normal designers as well as interface designers. Using these eight guidelines, it is possible to differentiate a good interface design from a bad one. These are beneficial in experimental assessment of identifying better GUIs.

Norman's Seven Principles

To assess the interaction between human and computers, Donald Norman in 1988 proposed seven principles. He proposed the seven stages that can be used to transform difficult tasks. Following are the seven principles of Norman −

- Use both knowledge in world & knowledge in the head.
- Simplify task structures.
- Make things visible.
- Get the mapping right (User mental model = Conceptual model = Designed model).
- Convert constrains into advantages (Physical constraints, Cultural constraints, Technological constraints).
- Design for Error.
- When all else fails − Standardize.

Heuristic Evaluation

Heuristics evaluation is a methodical procedure to check user interface for usability problems. Once a usability problem is detected in design, they are attended as an integral part of constant design processes. Heuristic evaluation method includes some usability principles such as Nielsen's ten Usability principles.

Nielsen's Ten Heuristic Principles

- Visibility of system status.
- Match between system and real world.
- User control and freedom.
- Consistency and standards.
- Error prevention.

- Recognition rather than Recall.
- Flexibility and efficiency of use.
- Aesthetic and minimalist design.
- Help, diagnosis and recovery from errors.
- Documentation and Help

The above mentioned ten principles of Nielsen serve as a checklist in evaluating and explaining problems for the heuristic evaluator while auditing an interface or a product.

Interface Design Guidelines

Some more important HCI design guidelines are presented in this section. General interaction, information

n display, and data entry are three categories of HCI design guidelines that are explained below.

General Interaction

Guidelines for general interaction are comprehensive advices that focus on general instructions such as −

- Be consistent.
- Offer significant feedback.
- Ask for authentication of any non-trivial critical action.
- Authorize easy reversal of most actions.
- Lessen the amount of information that must be remembered in between actions.
- Seek competence in dialogue, motion and thought.
- Excuse mistakes.
- Classify activities by function and establish screen geography accordingly.
- Deliver help services that are context sensitive.
- Use simple action verbs or short verb phrases to name commands.

Information Display

Information provided by the HCI should not be incomplete or unclear or else the application will not meet the requirements of the user. To provide better display, the following guidelines are prepared −

- Exhibit only that information that is applicable to the present context.
- Don't burden the user with data, use a presentation layout that allows rapid integration of information.
- Use standard labels, standard abbreviations and probable colors.
- Permit the user to maintain visual context.
- Generate meaningful error messages.
- Use upper and lower case, indentation and text grouping to aid in understanding.
- Use windows (if available) to classify different types of information.
- Use analog displays to characterize information that is more easily integrated with this form of representation.
- Consider the available geography of the display screen and use it efficiently.

Data Entry

The following guidelines focus on data entry that is another important aspect of HCI −

- Reduce the number of input actions required of the user.
- Uphold steadiness between information display and data input.
- Let the user customize the input.
- Interaction should be flexible but also tuned to the user's favored mode of input.
- Disable commands that are unsuitable in the context of current actions.

- Allow the user to control the interactive flow.
- Offer help to assist with all input actions.
- Remove "mickey mouse" input.

User Interface Design Principles

The dynamic characteristics of a system are described in terms of the dialogue requirements contained in seven principles of part 10 of the ergonomics standard, the ISO 9241. This standard establishes a framework of ergonomic "principles" for the dialogue techniques with high-level definitions and illustrative applications and examples of the principles. The principles of the dialogue represent the dynamic aspects of the interface and can be mostly regarded as the "feel" of the interface. The seven dialogue principles are:

- Suitability for the task: the dialogue is suitable for a task when it supports the user in the effective and efficient completion of the task.
- Self-descriptiveness: the dialogue is self-descriptive when each dialogue step is immediately comprehensible through feedback from the system or is explained to the user on request.
- Controllability: the dialogue is controllable when the user is able to initiate and control the direction and pace of the interaction until the point at which the goal has been met.
- Conformity with user expectations: the dialogue conforms with user expectations when it is consistent and corresponds to the user characteristics, such as task knowledge, education, experience, and to commonly accepted conventions.
- Error tolerance: the dialogue is error tolerant if despite evident errors in input, the intended result may be achieved with either no or minimal action by the user.
- Suitability for individualization: the dialogue is capable of individualization when the interface software can be modified to suit the task needs, individual preferences, and skills of the user.
- Suitability for learning: the dialogue is suitable for learning when it supports and guides the user in learning to use the system.

The concept of usability is defined of the ISO 9241 standard by effectiveness, efficiency, and satisfaction of the user. Part 11 gives the following definition of usability:

- Usability is measured by the extent to which the intended goals of use of the overall system are achieved (effectiveness).
- The resources that have to be expended to achieve the intended goals (efficiency).
- The extent to which the user finds the overall system acceptable (satisfaction).

Effectiveness, efficiency, and satisfaction can be seen as quality factors of usability. To evaluate these factors, they need to be decomposed into sub-factors, and finally, into usability measures.

The information presentation is described in Part 12 of the ISO 9241 standard for the organization of information (arrangement, alignment, grouping, labels, location), for the display of graphical objects, and for the coding of information (abbreviation, color, size, shape, visual cues) by seven attributes. The "attributes of presented information" represent the static aspects of the interface and can be generally regarded as the "look" of the interface. The attributes are detailed in the recommendations given in the standard. Each of the recommendations supports one or more of the seven attributes. The seven presentation attributes are:

- Clarity: the information content is conveyed quickly and accurately.
- Discriminability: the displayed information can be distinguished accurately.
- Conciseness: users are not overloaded with extraneous information.
- Consistency: a unique design, conformity with user's expectation.

- Detectability: the user's attention is directed towards information required.
- Legibility: information is easy to read.
- Comprehensibility: the meaning is clearly understandable, unambiguous, interpretable, and recognizable.

The user guidance in Part 13 of the ISO 9241 standard describes that the user guidance information should be readily distinguishable from other displayed information and should be specific for the current context of use. User guidance can be given by the following five means:

- Prompts indicating explicitly (specific prompts) or implicitly (generic prompts) that the system is available for input.
- Feedback informing about the user's input timely, perceptible, and non-intrusive.
- Status information indicating the continuing state of the application, the system's hardware and software components, and the user's activities.
- Error management including error prevention, error correction, user support for error management, and error messages.
- On-line help for system-initiated and user initiated requests with specific information for the current context of use.

Definition of Usability

Usability is a **quality attribute** that assesses how easy user interfaces are to use. The word "usability" also refers to methods for improving ease-of-use during the design process.

Usability is defined by **5 quality components**:

- **Learnability**: How easy is it for users to accomplish basic tasks the first time they encounter the design?
- **Efficiency**: Once users have learned the design, how quickly can they perform tasks?
- **Memorability**: When users return to the design after a period of not using it, how easily can they reestablish proficiency?
- **Errors**: How many errors do users make, how severe are these errors, and how easily can they recover from the errors?
- **Satisfaction**: How pleasant is it to use the design?

There are many other important quality attributes. A key one is **utility**, which refers to the design's functionality: Does it do what users need?

Usability and utility are equally important and together determine whether something is useful: It matters little that something is easy if it's not what you want. It's also no good if the system can hypothetically do what you want, but you can't make it happen because the user interface is too difficult. To study a design's utility, you can use the same user research methods that improve usability.

- Definition of **Utility** = whether it provides the **features you need**.
- Definition of **Usability** = how **easy & pleasant** these features are to use.
- Definition of **Useful** = **usability + utility**.

Why Usability Is Important

On the Web, usability is a necessary condition for survival. If a website is difficult to use, people **leave**. If the homepage fails to clearly state what a company offers and what users can do on the site, people **leave**. If users get lost on a website, they **leave**. If a website's information is hard to read or doesn't answer users' key questions, they **leave**. Note a pattern here? There's no such thing as a user reading a website manual or otherwise spending much time trying to figure out an interface. There are plenty of other websites available; leaving is the first line of defense when users encounter a difficulty.

The first law of ecommerce is that if users cannot *find* the product, they cannot *buy* it either.
For **intranets**, usability is a matter of employee productivity. Time users waste being lost on your intranet or pondering difficult instructions is money you waste by paying them to be at work without getting work done.
Current best practices call for spending about **10% of a design project's budget** on usability. On average, this will more than double a website's desired quality metrics (yielding an improvement score of 2.6) and slightly less than double an intranet's quality metrics. For software and physical products, the improvements are typically smaller — but still substantial — when you emphasize usability in the design process.
For internal design projects, think of doubling usability as cutting training budgets in half and doubling the number of transactions employees perform per hour. For external designs, think of doubling sales, doubling the number of registered users or customer leads, or doubling whatever other KPI (key performance indicator) motivated your design project.

How to Improve Usability

There are many methods for studying usability, but the most basic and useful is **user testing**, which has 3 components:

- Get hold of some **representative users**, such as customers for an ecommerce site or employees for an intranet (in the latter case, they should work outside your department).
- Ask the users to perform **representative tasks** with the design.
- **Observe** what the users do, where they succeed, and where they have difficulties with the user interface. Shut up and let the users do the talking.

It's important to test users individually and let them solve any problems on their own. If you help them or direct their attention to any particular part of the screen, you have contaminated the test results.
To identify a design's most important usability problems, testing 5 users is typically enough. Rather than run a big, expensive study, it's a better use of resources to run many small tests and revise the design between each one so you can fix the usability flaws as you identify them. Iterative design is the best way to increase the quality of user experience. The more versions and interface ideas you test with users, the better.
User testing is different from focus groups, which are a poor way of evaluating design usability. Focus groups have a place in market research, but to evaluate interaction designs you must closely observe individual users as they perform tasks with the user interface.

Usability Design and Heuristics

The design and usability of these systems leaves an effect on the quality of people's relationship to technology. Web applications, games, embedded devices, etc., are all a part of this system, which has become an integral part of our lives. Let us now discuss on some major components of this system.

Concept of Usability Engineering

Usability Engineering is a method in the progress of software and systems, which includes user contribution from the inception of the process and assures the effectiveness of the product through the use of a usability requirement and metrics.
It thus refers to the *Usability Function* features of the entire process of abstracting, implementing & testing hardware and software products. Requirements gathering stage to installation, marketing and testing of products, all fall in this process.

Goals of Usability Engineering

- Effective to use − Functional

- Efficient to use − Efficient
- Error free in use − Safe
- Easy to use − Friendly
- Enjoyable in use − Delightful Experience

Usability

Usability has three components − effectiveness, efficiency and satisfaction, using which, users accomplish their goals in particular environments. Let us look in brief about these components.

- **Effectiveness** − The completeness with which users achieve their goals.
- **Efficiency** − The competence used in using the resources to effectively achieve the goals.
- **Satisfaction** − The ease of the work system to its users.

Usability Study

The methodical study on the interaction between people, products, and environment based on experimental assessment. Example: Psychology, Behavioral Science, etc.

Usability Testing

The scientific evaluation of the stated usability parameters as per the user's requirements, competences, prospects, safety and satisfaction is known as usability testing.

Acceptance Testing

Acceptance testing also known as User Acceptance Testing (UAT), is a testing procedure that is performed by the users as a final checkpoint before signing off from a vendor. Let us take an example of the handheld barcode scanner.

Let us assume that a supermarket has bought barcode scanners from a vendor. The supermarket gathers a team of counter employees and make them test the device in a mock store setting. By this procedure, the users would determine if the product is acceptable for their needs. It is required that the user acceptance testing "pass" before they receive the final product from the vendor.

Software Tools

A software tool is a programmatic software used to create, maintain, or otherwise support other programs and applications. Some of the commonly used software tools in HCI are as follows −

- **Specification Methods** − The methods used to specify the GUI. Even though these are lengthy and ambiguous methods, they are easy to understand.
- **Grammars** − Written Instructions or Expressions that a program would understand. They provide confirmations for completeness and correctness.
- **Transition Diagram** − Set of nodes and links that can be displayed in text, link frequency, state diagram, etc. They are difficult in evaluating usability, visibility, modularity and synchronization.
- **Statecharts** − Chart methods developed for simultaneous user activities and external actions. They provide link-specification with interface building tools.
- **Interface Building Tools** − Design methods that help in designing command languages, data-entry structures, and widgets.
- **Interface Mockup Tools** − Tools to develop a quick sketch of GUI. E.g., Microsoft Visio, Visual Studio .Net, etc.
- **Software Engineering Tools** − Extensive programming tools to provide user interface management system.
- **Evaluation Tools** − Tools to evaluate the correctness and completeness of programs.

Software Development Life Cycle

Software Development Life Cycle, SDLC for short, is a well-defined, structured sequence of stages in software engineering to develop the intended software product.

SDLC provides a series of steps to be followed to design and develop a software product efficiently

Software is more than just a program code. A program is an executable code, which serves some computational purpose. Software is considered to be collection of executable programming code, associated libraries and documentations. Software, when made for a specific requirement is called **software product.**

Engineering on the other hand, is all about developing products, using well-defined, scientific principles and methods.

Software engineering is an engineering branch associated with development of software product using well-defined scientific principles, methods and procedures. The outcome of software engineering is an efficient and reliable software product.

Definitions

IEEE defines software engineering as:

(1) The application of a systematic,disciplined,quantifiable approach to the development,operation and maintenance of software; that is, the application of engineering to software.

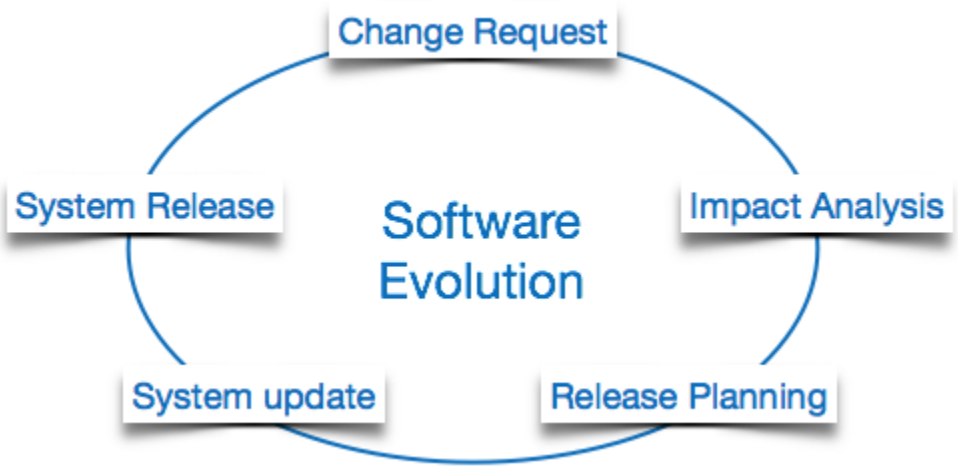
(2) The study of approaches as in the above statement.

Fritz Bauer, a German computer scientist, defines software engineering as:

Software engineering is the establishment and use of sound engineering principles in order to obtain economically software that is reliable and work efficiently on real machines.

Software Evolution

The process of developing a software product using software engineering principles and methods is referred to as **software evolution.** This includes the initial development of software and its maintenance and updates, till desired software product is developed, which satisfies the expected requirements.



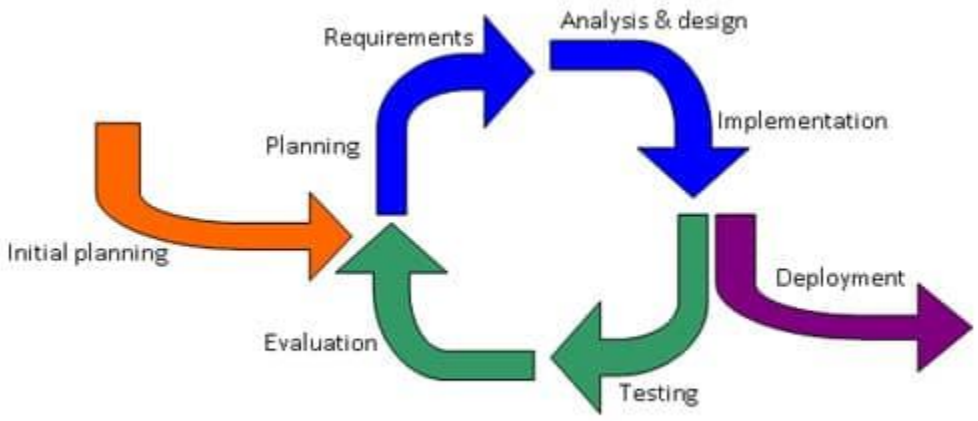
Evolution starts from the requirement gathering process. After which developers create a prototype of the intended software and show it to the users to get their feedback at the early stage of software product development. The users suggest changes, on which several consecutive updates and maintenance keep on changing too. This process changes to the original software, till the desired software is accomplished.

Even after the user has desired software in hand, the advancing technology and the changing requirements force the software product to change accordingly. Re-creating software from scratch and to go one-on-one with requirement is not feasible. The only feasible and economical solution is to update the existing software so that it matches the latest requirements.

Iterative Model

Updated: July 2, 2017 - Amir Ghahrai

What is the Iterative Model?



Model 1: Typical iterative development process

iterative-model

An iterative life cycle model does not attempt to start with a full specification of requirements. Instead, development begins by specifying and implementing just part of the software, which can then be reviewed in order to identify further requirements. This process is then repeated, producing a new version of the software for each cycle of the model.

Consider an iterative life cycle model which consists of repeating the following four phases in sequence:

A Requirements phase, in which the requirements for the software are gathered and analysed. Iteration should eventually result in a requirements phase that produces a complete and final specification of requirements.

A Design phase, in which a software solution to meet the requirements is designed. This may be a new design, or an extension of an earlier design.

An Implementation and Test phase, when the software is coded, integrated and tested.

A Review phase, in which the software is evaluated, the current requirements are reviewed, and changes and additions to requirements proposed.

For each cycle of the model, a decision has to be made as to whether the software produced by the cycle will be discarded, or kept as a starting point for the next cycle (sometimes referred to as **incremental prototyping**). Eventually a point will be reached where the requirements are

complete and the software can be delivered, or it becomes impossible to enhance the software as required, and a fresh start has to be made.

The iterative life cycle model can be likened to producing software by successive approximation. Drawing an analogy with mathematical methods that use successive approximation to arrive at a final solution, the benefit of such methods depends on how rapidly they converge on a solution.

The key to successful use of an iterative software development life cycle is rigorous validation of requirements, and verification (including testing) of each version of the software against those requirements within each cycle of the model.

The first three phases of the example iterative model is in fact an abbreviated form of a sequential **V Model** or **waterfall Model** of development. Each cycle of the model produces software that requires testing at the unit level, for software integration, for system integration and for acceptance. As the software evolves through successive cycles, tests have to be repeated and extended to verify each version of the software.

Advantages of Iterative Model

- Generates working software quickly and early during the software life cycle.
- More flexible – less costly to change scope and requirements.
- Easier to test and debug during a smaller iteration.
- Easier to manage risk because risky pieces are identified and handled during its iteration.
- Each iteration is an easily managed milestone.

Disadvantages of Iterative Model

- Each phase of an iteration is rigid and do not overlap each other.
- Problems may arise pertaining to system architecture because not all requirements are gathered up front for the entire software life cycle

Prototype Model

The Software Prototyping refers to building software application prototypes which displays the functionality of the product under development, but may not actually hold the exact logic of the original software.

Software prototyping is becoming very popular as a software development model, as it enables to understand customer requirements at an early stage of development. It helps get valuable

feedback from the customer and helps software designers and developers understand about what exactly is expected from the product under development.

What is Software Prototyping?

Prototype is a working model of software with some limited functionality. The prototype does not always hold the exact logic used in the actual software application and is an extra effort to be considered under effort estimation.

Prototyping is used to allow the users evaluate developer proposals and try them out before implementation. It also helps understand the requirements which are user specific and may not have been considered by the developer during product design.

Following is a stepwise approach explained to design a software prototype.

- **Basic Requirement Identification**

This step involves understanding the very basics product requirements especially in terms of user interface. The more intricate details of the internal design and external aspects like performance and security can be ignored at this stage.

- **Developing the initial Prototype**

The initial Prototype is developed in this stage, where the very basic requirements are showcased and user interfaces are provided. These features may not exactly work in the same manner internally in the actual software developed. While, the workarounds are used to give the same look and feel to the customer in the prototype developed.

- **Review of the Prototype**

The prototype developed is then presented to the customer and the other important stakeholders in the project. The feedback is collected in an organized manner and used for further enhancements in the product under development.

- **Revise and Enhance the Prototype**

The feedback and the review comments are discussed during this stage and some negotiations happen with the customer based on factors like – time and budget constraints and technical feasibility of the actual implementation. The changes accepted are again incorporated in the new Prototype developed and the cycle repeats until the customer expectations are met.

Prototypes can have horizontal or vertical dimensions. A Horizontal prototype displays the user interface for the product and gives a broader view of the entire system, without concentrating on internal functions. A Vertical prototype on the other side is a detailed elaboration of a specific function or a sub system in the product.

The purpose of both horizontal and vertical prototype is different. Horizontal prototypes are used to get more information on the user interface level and the business requirements. It can even be presented in the sales demos to get business in the market. Vertical prototypes are technical in nature and are used to get details of the exact functioning of the sub systems. For example, database requirements, interaction and data processing loads in a given sub system.

Software Prototyping - Types

There are different types of software prototypes used in the industry. Following are the major software prototyping types used widely −

- **Throwaway/Rapid Prototyping**

Throwaway prototyping is also called as rapid or close ended prototyping. This type of prototyping uses very little efforts with minimum requirement analysis to build a prototype. Once the actual requirements are understood, the prototype is discarded and the actual system is developed with a much clear understanding of user requirements.

- **Evolutionary Prototyping**

Evolutionary prototyping also called as breadboard prototyping is based on building actual functional prototypes with minimal functionality in the beginning. The prototype developed forms the heart of the future prototypes on top of which the entire system is built. By using evolutionary prototyping, the well-understood requirements are included in the prototype and the requirements are added as and when they are understood.

- **Incremental Prototyping**

Incremental prototyping refers to building multiple functional prototypes of the various sub-systems and then integrating all the available prototypes to form a complete system.

- **Extreme Prototyping**

Extreme prototyping is used in the web development domain. It consists of three sequential phases. First, a basic prototype with all the existing pages is presented in the HTML format. Then the data processing is simulated using a prototype services layer. Finally, the services are implemented and integrated to the final prototype. This process is called Extreme Prototyping used to draw attention to the second phase of the process, where a fully functional UI is developed with very little regard to the actual services.

- **Software Prototyping - Application**

Software Prototyping is most useful in development of systems having high level of user interactions such as online systems. Systems which need users to fill out forms or go through various screens before data is processed can use prototyping very effectively to give the exact look and feel even before the actual software is developed.

Software that involves too much of data processing and most of the functionality is internal with very little user interface does not usually benefit from prototyping. Prototype development could be an extra overhead in such projects and may need lot of extra efforts.

Software Prototyping - Pros and Cons

Software prototyping is used in typical cases and the decision should be taken very carefully so that the efforts spent in building the prototype add considerable value to the final software developed. The model has its own pros and cons discussed as follows.

The advantages of the Prototyping Model are as follows −

- Increased user involvement in the product even before its implementation.
- Since a working model of the system is displayed, the users get a better understanding of the system being developed.
- Reduces time and cost as the defects can be detected much earlier.
- Quicker user feedback is available leading to better solutions.

- Missing functionality can be identified easily.
- Confusing or difficult functions can be identified.

The Disadvantages of the Prototyping Model are as follows −

- Risk of insufficient requirement analysis owing to too much dependency on the prototype.
- Users may get confused in the prototypes and actual systems.
- Practically, this methodology may increase the complexity of the system as scope of the system may expand beyond original plans.
- Developers may try to reuse the existing prototypes to build the actual system, even when it is not technically feasible.
- The effort invested in building prototypes may be too much if it is not monitored properly.

Introduction: Evaluation in User Centered Design Processes

Evaluation is an important part of any user-centered design process, where it is used to:

1. To identify usability problems.
2. To assess whether the GUI design satisfies usability requirements.
3. To evaluate whether the GUI design will be usable in practice by its intended users.

There are both formative and summative aspects to evaluation.
1 is carried out throughout the process to help form the design.
2 and 3 occur towards the end, and become summative in that they assess the success of the whole design exercise.

Methods and Techniques

- analytic evaluation
- expert evaluation
- observational evaluation
- survey evaluation
- experimental evaluation

  Analytic Evaluation

  A paper-based analysis of a definition of the user interface, either sketched, in Natural Language or some semi-formal language, e.g. Command Grammar Language or GOMS

  GOOD

  - no need to build prototype
  - no need to arrange user testing

  BAD

  - can be very time consuming
  - often require specialist psychological knowledge
  - don't tell us anything about errors or learning behaviour

Fuller details

Expert Evaluation

Not quite empirical research
Expert mustn't be part of the design team
Need interface description, task description, user model
Structured vs. unstructured reporting
Predefined categorizations, e.g. Heuristic Evaluation
GOOD

- efficient, quick, rich source of comments
- often source of solutions as well as problems

BAD

- experts have biases
- experts are not real users

Fuller details

Observational Evaluation

Observing user behavior in lab or workplace setting
Direct observation (but Hawthorne effect)
Video recording with playback, participative or not
Verbal protocols: think aloud, question asking
Software logging
GOOD

- real users

BAD

- interference with performance
- post-task rationalization
- often labour intensive

Example: anywhere where evidence is needed of e.g. recall of commands, planning, understanding of operations, messages, level of performance of system and user, etc.

Fuller details

Survey Evaluation

Interviews

Structured and sequenced form filling
Unstructured, topics to cover, but no fixed sequence
GOOD

- can obtain in-depth response from user
- can enable new issues to emerge

BAD

- time consuming (expensive)

- requires training and skill to carry out

Questionnaires

Open questions
Closed questions --> rating scale
Checklists
Multipoint scales, including Likert Scale
Semantic differential scale
Ranked order
GOOD

- cheap to administer to a large number of users (but need to motivate user to return it)
- easy to analyse - unless unstructured responses are allowed

BAD

- time and skill required to develop the questionnaire (or commercial set can be used - expense)
- will only uncover what is looked for

Fuller details

Experimental Evaluation

Testing of hypothesis
Variables
Constants
Measure
Can the hypothesis be stated in a way that can be tested?
Statistical analysis to check reliability of results
Pilot studies
GOOD

- reliable results

BAD

- need specialist knowledge
- resources needed to set up experiment
- can't be used for every design decision

References

Redmond-Pyle, David & Moore, Alan Graphical User Interface Design and Evaluation: A Practical Process Prentice Hall 1995