Task-oriented System Engineering

Avigdor Zonnenshain

Rafael

P.O.Box 2250, Haifa 31021, Israel
Tel: +972 52 289 1773
avigdorz@rafael.co.il

Avi Harel

Ergolight Ltd.

6 Givon Str., Haifa 34335, Israel
Tel: +972 54 453 4501,
ergolight@gmail.com

Abstract. Traditional system engineering (SE) is technology driven. In the design of interactive systems we typically assume that the human operator traces the changes in the system states and operates the system correctly. However, the human operator obeys different rules. The difference between the designers' and the operator's rules often results in loss of productivity, decreased performance and accidents, and eventually, loss of market share. In special cases, it makes sense to demand that the user follows the designers' logic. However, in most projects, it is the designer who needs to adapt to the user's logic. The article analyzes the limitations of common design practices in resolving mismatches between the system and the user states. Complex system engineering (CSE) is a framework suitable for handling the human operator as a critical system component. This article presents a methodology for handling the human attributes in this framework. The methodology extends the capability of common practices of user-centered design and usability testing, which miss critical interdisciplinary issues. To resolve state mismatches, the system engineers must be aware of the user's logic. Such knowledge should be reflected in the system architecture, to ensure that the user interface provides protection against unexpected user events and to facilitate the system operation in new situations, such as in emergency. The methods and guidelines presented here are applicable to the whole development cycle.

Technology-driven System Engineering

The logical gap. Traditional system engineering is technology driven. In the design of interactive systems we typically assume that the human operator traces the changes in the system states and operates the system correctly. However, the human operator obeys different rules (e.g., http://www.aesthetic-images.com/ebuie/usability_semantics.html). The gap between the designers' and the operator's logic often results in loss of productivity, decreased performance and accidents. Eventually, this might end up in losing a market share. We start with examples of the risks of disregarding the user's logic.

Example: cable TV. A remote control of a cable TV system enables turning on and off and changing the channels of both the TV set and the cable converter. A special button enables the users to set to either TV or converter mode. The design seems logical and easy to understand. Many users are willing to learn and follow this logic. Many others find it confusing. They often forget to select the proper mode, resulting in them unintentionally turning the converter off or

setting the TV set to a wrong channel. They call customer support and report seeing snow on the screen.

This example demonstrates the well known problem of **system-user state mismatch**. The designer or the remote control unit assumed that the user will trace and notice the changes in the TV/Converter modes of the remote control. The users, however, act carelessly, assuming that the mode is according to their intentions. The design mistake is of assuming that the users behave according to the designers' logic.

Traditional SE practices, as well as common usability engineering practices do not include guidelines for avoiding mode dependent functions. Such guidelines could prevent or at least reduce the chance for this kind of mode mismatch. Also, neither traditional SE practices nor common usability engineering practices include guidelines for how to ensure that the users are aware of the active mode in case of mode mismatch, and that they know how to resolve such situations. This well known problem is orphan. Moreover, regular usability testing, based on prototypes, is not very effective for identifying this kind of problems. It enables to detect situations of negative user experience, but not to identify the state mismatch causing the negative user experience. Special kind of usability tests of the user's behaviour are required to identify state mismatch, involving tracing both the system states and the user's perception of the system states.

The methodology introduced here includes methods and guidelines for preventing situations of state mismatch, by reducing functional overload of user controls or by intelligent control allocation. Also, the methodology incorporates methods for detecting and protecting from unexpected user events by software, and for supporting the user in recovery procedures (Zonnenshein & Harel, 2008).

Example: machine damage. A production line was designed to operate with coolant valve open. Due to a transient power failure of the controller, the controller invoked an unusual command sequence, and the production line started with the coolant valve closed.

This example demonstrates the problem of an **intra-system state mismatch** due to procedural disorder. Both the controller and the unit worked according to the specifications. However, they were not coordinated. The designers wrongly assumed that the system will never deviate from the specified command sequence. The system was not designed to detect and protect from deviations from this sequence.

Traditional SE practices may include guidelines for detecting such mismatches, but they do not incorporate guidelines for notifying the operator about them. Neither traditional SE practices nor common usability engineering practices provide guidelines for instructing the operator about the recovery procedures.

The methodology introduced here includes methods and guidelines for detecting intra-system state mismatches and for guiding the users in the recovery procedures (Zonnenshein & Harel, 2008).

Example: emergency operation. On March 28, 1979, the main feedback pumps in the secondary cooling system of the Three Miles Island nuclear plant failed. Due to a design mistake, a pressure valve did not close after being open, and the reactor became overheated. Due to design mistakes, important indicators were missing. The scope of the accident became clear over the course of five days, as a number of agencies at the local, state and federal levels tried to diagnose the problem and decide whether the on-going accident required a full emergency evacuation of the local community, if not the entire area to the west/southwest. There is consensus that the accident was exacerbated by wrong decisions made because the operators

were overwhelmed with information, much of it irrelevant, misleading or incorrect. (http://en.wikipedia.org/wiki/Three_Mile_Island_accident).

This example demonstrates the problem of **state ambiguity**. The signals about the problematic system situation did not indicate the cause for the exceptional values that the sensors measured.

Technology driven SE practices, suited to the designer's logic, assume that the users can investigate the sources for the exceptional situations, because they must know the rules. This assumption is based on wishful thinking.

Troubleshooting is an interdisciplinary activity, intended to map the system situation into the operator's mind. The methodology introduced here includes methods and guidelines for automatic troubleshooting (Zonnenshein & Harel, 2008).

Traditional SE ignores usability. Typically, developers (e.g., programmers) who are fond of technology are careless about user needs (Weinberg, 1971). Common SE practices do not target the usability requirements. Too many products fail due to usability limitations after being qualified by formal QA procedures: because they are useless, because they are too complex to use, because the enable critical user errors. Too many systems intended to protect home security installations work perfectly at the QA qualification stage, but fail when they are needed, because psychological aspects were not considered at the design phase. Too much time we waste trying to find out why the software behaves so strangely, or what should we do in order that the TV will show a picture other than that of falling snow.

Blaming the users. It is typical to technology driven engineers to blame the users for not following the designers' instructions or intentions, for behaving illogically. Often, we can hear frustrated engineers suggesting that the customer should change the user. This approach is convenient for the developers. However, blaming the users often ends up in overlooking the reasons for the user's errors, enabling the users to repeat the errors. We cannot change the user. However, we can change the design. It is the system manufacturers' responsibility to prevent user errors, and if the design is error prone, it is the developer who should be blamed (e.g. http://www.efluxmedia.com/news_Whos_To_Blame_In_Deadly_Train_Collision_25718.html).

Interdisciplinary problem. It is typical to technology driven engineers to focus on technological aspects of the system and to let somebody else consider the human factors. The problem with this approach is that interaction failures are due to tight interdisciplinary coupling. For example, in order to ensure seamless operation of a cable TV, the system engineer has to be aware of the drawbacks of functional control overload, and to require that the usability engineer will find ways to prevent state mismatch. Also, in order to prevent machine damage as in the second example, the system engineer needs to inform the usability engineers about all known failure modes, as well as about recovery procedures. And, in order to prevent accidents such as that of the TMI nuclear plant, the usability engineer should be informed about of all the possible system failures, and the means provided to identify them.

Blaming the system engineers. In order to prevent similar mishaps, somebody who knows how the system might fail needs to communicate the knowledge with the user interface designers. Typically, it is the system engineer who is in charge of preventing and mitigating the risks of all sources of system failures, including those generated by the human operator. If the user is liable to make a mistake, and the means to mitigating the risks of such mistakes are known, then it is the system engineer who should be blamed for the resulting mishaps.

Feature-oriented engineering. Disciplined system development begins with task analysis. We formalize the user's goals and we break them down to minitasks. Next, we implement the minitasks. Each minitask is transformed into a feature. Technology driven engineering is feature oriented. This means that at this point we stop thinking about the user tasks. We deal only with features derived from the minitasks. Now, it is the user's responsibility to get the proper feature at each stage of the interaction.

State compatibility. In many practical systems, many of the features provided behave differently in different system states. When a state-dependent feature is invoked, the user needs to be aware of the state: to make sure that the system is in the proper state, enabling the desired feature variant. Otherwise, if the user fails to trace the system state or to verify that the system is in the proper state, the wrong feature will be actuated.

State-dependent features provide many opportunities for proving the validity of Murphy's law (http://en.wikipedia.org/wiki/Murphys_law). Most operational failures attributed to user errors are due to the user's failure to verify that the system is in the proper state. Many examples of use system mismatch may be found in http://www.ergolight-sw.com/CHI/Company/Articles/ESE-Incose2008-P192.pdf

The operational database. The operational database consists of the system states and the operational procedures. Formally, an operational procedure may be represented as a directed graph, in which the nodes represent state dependency and the arcs represent the system features. Each branch in the directed graph represents an operational scenario, namely, a sequence of features, conditioned by the system states.

At design time, the designers of the operational procedures define a limited set of system states. It is assumed that in run time, the users may know, remember and recognize the system states. This assumption is valid only for very simple systems. In practice, even the system designers do not remember all the details about the state changes and the state dependency.

At test time, after suitable training, the testers of the user interface may manage to verify that they operate in the proper system state.

At run time, the human operators typically know the details required for executing main tasks, namely, tasks that they need to repeat frequently. However, unlike the testers, the users access only a small subset of the operational database. For example, because emergency situations are rare, the users do not have a chance to exercise the emergency procedures beforehand. Unless the users have a special training program, they cannot remember all the details required to operate the emergency procedures successfully.

The operational context. The testers of the user interface operate in well defined scenarios, in which they manage to trace the state changes, and they can take the time required to verify that the system state suits their intention. This is not true for run-time operation. Besides the system operation, the human operators are typically engaged in many other tasks. Consequently, they might miss opportunities to perceive changes in the system state. Being busy doing other tasks, they might also forget that they need to check the system state. Consequently, they make mistakes.

The user's logic. It is too easy to blame the users for being illogical. They are, but their logic is different from that of the designers. Logical reasoning depends on data and on rules. The logical gap is due to differences in the operational and context databases, and in the method used employing the rules in decision making. Suppose that the user does not recall the condition for a particular procedure step. Unlike system testers, who cannot complete the test procedure until

they believe they know the condition and its effect, the system operator needs to decide based on partial information. Testers need to follow deterministic logic. They need to follow the rules, whether they are documented or elicited from the designers. On the contrary, run-time operators need to apply fuzzy logic. They do not have designers around them to enquire about state recognition and state dependency, to help them identify the current system state and to instruct them how to proceed. They may have already searched the operational instructions, or they already know that not all the details are documented in the manual or the Help system. Yet, they need to decide, based on partial information. They do not have the privilege of testers. Often, under time stress, they need to gamble. Ad-hoc, in accident investigation, if we wrongly assume that they know and remember the operational procedures, their decisions often might seem illogical.

Limitations of Technology-driven SE

Common SE practices fail to prevent mishaps as those described above. The reason for this is that common SE practices persistently ignore the most critical system component, namely, the human operator (Case, 1997). The bottleneck for achieving high levels of productivity and safety is the human operators, who often fail to follow the designers' instructions and expectations. In order to deal with this bottleneck, we need to include the human operator in the system model. We need to consider the properties of the user and to take care of the user's failure modes.

The International Council on Systems Engineering (INCOSE) defines system engineering as

 " an interdisciplinary approach and means to enable the realization of successful systems."

This definition suggests that usability, being a discipline required for the realization of successful systems, is part of system engineering. Still, common SE practices fail to prevent mishaps as those described above. Why? Because it is common practice to assume that the run-time user will operate the system according to the designers' expectations. It is common practice to focus on technology, ignoring cognitive aspects of the interaction.

Sources of the usability gap. Everybody in the system development team expects that it will be usable. Yet, it rarely happens. Berkun (http://www.scottberkun.com/essays/22-the-list-of-reasons-ease-of-use-doesnt-happen-on-engineering-projects/) provided a list for why systems are not always easy to use. This section provides an overview of the forces within the system development team that act against usability, and proposes that customer utility should be set as the main goal. It should be commented that naturally, many system engineers deny the kind of critics that this section might hinder (one of the reviewers commented that the article overuses cartoons as sources).
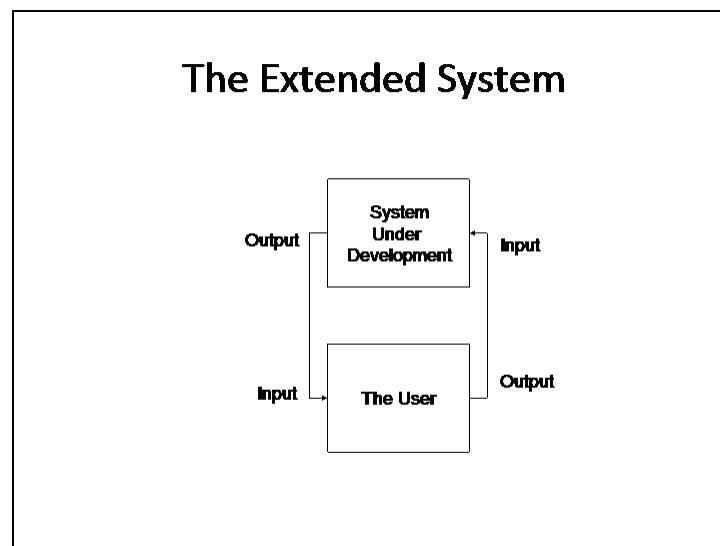
   The developer's intuition. "An intuitive interface asks no more of the user than what they either already know, or can immediately deduce from previous life experience. Implied is that intuition is wisdom assumed and shared within a community — the community of users familiar with the task and with the environment in which it is performed" ([Buie and Vallone, 1997](#)). The usability problem results from the developers' intuition, that of highly skilled users, being applied to regular users, who are not familiar with the system behavior ([Martin cartoon: http://www.nevtron.si/borderline/archive2/intuiti.gif](http://www.nevtron.si/borderline/archive2/intuiti.gif)). After getting used to the prototype, developers typically judge the system behavior as experienced users (e.g., [Dilbert cartoon: http://web.mit.edu/is/usability/IAP/2003/Session1/Images/ctrl-alt.gif](http://web.mit.edu/is/usability/IAP/2003/Session1/Images/ctrl-alt.gif)). For them, the system behavior is obvious, and they fail to understand why a user, who sees a certain feature for the

first time, would not realize what it should do, and how (e.g. [Dilbert cartoon: http://web.mit.edu/is/usability/IAP/2003/Session1/Images/Stupid-users.gif](http://web.mit.edu/is/usability/IAP/2003/Session1/Images/Stupid-users.gif)).

Designers Creativity. UI designers do not always promote usability: simple UI appearance, easy look and feel might often be boring for some designers. For example, website designers love to apply flash technology, which is 99% bad ([Nielsen alertbox: http://www.useit.com/alertbox/20001029.html](http://www.useit.com/alertbox/20001029.html)).

Complex System Engineering

The Extended System. The extended system describes the customers' view of the system. Typically, the customers may be interested in technical features and in functional features, such as performance and reliability, but eventually, they need to know if the users can complete their tasks in time, how reliably they do their jobs and what are the safety levels involved. Therefore, besides the system under development, the extended system includes also the user and the user interaction with the system, as demonstrated in the following chart:

Typically, the user is a critical intelligent, self-adaptive element of the extended system, which suggests that we impose the framework of CSE on the extended system.

We apply here the framework of CSE, in which the user is a critical system component. CSE processes and methods should be applied when the system to be developed is associated with the following:

- Cognitive, self-adapting elements (e.g., humans or very smart computers) are present within the system.
- Emergent behavior is dominant and will significantly influence the system's performance and effectiveness (this is often a direct result of the previous point)
- Elements of the system are added, removed, or functionally modified during the scenario (human operators often do this)
- The environment and interfacing systems will change and are not completely known at the time of development
- The system development and its funding are not under a single, central authority. Many change agents are at work.

- The system is sufficiently complex, so that exhaustive testing of all possible combinations of inputs and all possible human operator interactions is not feasible.

The original framework of CSE deals with the properties common to users and intelligent sub systems (e.g., Oliver et al., 1997). Instead, few techniques for interaction definition, such as protocol definition and methods for error detection, are similar. Also, few of the most powerful principles and means employed for usability assurance are also applicable to secure the interaction between any elements of any complex system, regardless of their intelligence. On the other hand, there are specific characteristics of the human operators, such as the perception of warning signals, which require special treatment.

Our approach emphasizes that the users' logic is different from that of intelligent sub systems, which is derived from that of the designers. Intelligent systems will always behave as engineers instruct them to do. Typically, they are deterministic and their behavior seems logical. However, this logic is based on a limited database, namely, the knowledge that the engineers managed to formulate as rules. The behavior of the human operators is also logical, but their logic is unknown to the developers, for two reasons: First, because it relies on a huge database, which has been accumulated during years of experience in managing various kinds of situations. Second, because the users need to decide based on partial information, and therefore they need to gamble. These differences motivate the need for the methodology introduced here. In our approach, the design of interactive systems should consider the special features of the user's logic.

The Quality of Interactive Systems

How does the quality of interactive systems differ from that of automated systems?

The definition of quality of automated systems is built bottom up. We define performance, reliability, recovery costs of the system units, and we compute these attributes for the whole system using mathematical manipulations. The quality of interactive systems is defined in terms of the user tasks. To evaluate the impact of the human operator on the system utility, we need to evaluate the barriers to system efficiency and reliability. Research on Human Factors suggests that focusing on system performance and reliability is practically useless, unless we also take special care of the user performance and reliability:

- **Performance.** The time required for the operators to evaluate the system state and decide what to do next is typically higher by an order of magnitude than the system response time. Instead of measuring the system response time, we should measure the time elapsed from the moment the user decides to perform a task until its completion. Typically, most of the elapsed time is wasted because the user fails to follow the operational procedures, attempting to recover from unwanted system response to unexpected actions. SE should regard user productivity, rather than system performance (Landauer, 1993).
- **Reliability.** The operators MTBF is about 10% of the overall operation time, higher by several orders of magnitude than that of the system. Instead of measuring component failure rates, such as by MTBF, we should measure operational failure rates, such as the rate of almost-accidents due to user errors. This is especially true for safety-critical systems, in which the costs of an accident are much higher than those of maintenance. Operational reliability is the key to task performance. (Example: http://www.jnd.org/dn.mss/commentary_huma.html).
- **Recovery costs.** The operators' MTTR is about 50% of the overall operation time, higher by several orders of magnitude than that of the system. Instead of measuring maintenance costs,

such as by MTTR, we should measure the time it takes for the users to recover from system failures.

- **Logic.** An application that is logical in its internal design and produces accurate results may nevertheless be difficult to use. The reason for this is that logic is not absolute. It is subjective, it is task related, and it changes over time. Typically, it applies to the internals of the application. Therefore, the user has difficulty following the developer's logic. ([Buie and Vallone: http://www.aesthetic-images.com/ebuie/larger_vision.html](http://www.aesthetic-images.com/ebuie/larger_vision.html)).

Managing the risks of usability deficiencies

The method proposed here for usability assurance is based on the common methodology of risk management. Risk management is a structured approach to managing uncertainty related to a threat, a sequence of human activities including: risk assessment, strategies development to manage it, and mitigation of risk using managerial resources. (http://en.wikipedia.org/wiki/Risk_management#Risk_retention). The previous section demonstrates the risk assessment of usability defects. The remaining of this article discusses strategies for managing these risks, and the implied requirements for managerial resources.

Potential risk treatments. Once risks have been identified and assessed, all techniques to manage the risk fall into one or more of these four major categories (Dorfman, 2007):

- Avoidance (eliminate)
- Reduction (mitigate)
- Transference (outsource or insure)
- Retention (accept and budget)

Barriers to risk treatments. In order to apply the treatments we need to have the management support in adopting a new strategy, which often contradicts the traditional strategies:

   Marketing-oriented engineering. People often confuse usability with marketing. However, marketing needs often conflict with usability ([Dilbert cartoon: http://www.guuui.com/images/20030209.gif](http://www.guuui.com/images/20030209.gif)). The problem is that marketing follows the user's buying forces, which are different from their usability needs. For example, when applying banners in marketing campaigns, we intentionally distract the users from their original goals, in favor of the marketing goals. Marketing managers think of attracting potential customers, disregarding the actual customers ([Dilbert cartoon: http://www.idblog.org/images/dilbert6-1.gif](http://www.idblog.org/images/dilbert6-1.gif)). They encourage usage of gimmicks, such as splash screens, to highlight new features that sell, regardless of the facts that these gimmicks hamper seamless operation. Marketing forces are according to the customers' wills, which are different from the users' needs. For example, a key feature that ensures usability is simplicity. However, marketing managers encourage complexity (http://www.joelonsoftware.com/items/2006/12/09.html). Leading usability practitioners have already noticed that people are not willing to pay for a system that looks simpler, because it looks less capable. Even a fully automatic system should contain lots of buttons and knobs, to make it look powerful (http://www.jnd.org/dn.mss/simplicity_is_highly.html). Before using a product, people will judge its desirability and quality based on 'what it does' (i.e. the number of features). Even though they may be aware that usability is likely to suffer, they will mostly choose products with many features. After having used these products however, usability will start to matter more than features and people will choose easy-to-use products over products with many features. The dilemma is that in order to maximize initial sales one needs to build products with many features, products that do lots of "stuff". But in order to maximize repeat sales,

customer satisfaction and retention, one needs to prioritize ease-of-use over features (http://www.lukew.com/ff/entry.asp?433).

Customer-oriented engineering. By disregarding usability, marketing managers often encourage developing systems that are difficult to use. Sometimes, however, they are right in doing so, because they do what the customers want, which is often not what they need. How should we balance usability against marketing? How can we conclude which of the two factors is more significant? The answer depends on the utility for the customers. However, even when marketing is considered more important, usability should be considered. For example, suppose that in order that the system looks powerful, the customers demand many features, and that all of them are apparent and easy to access. Still, usability engineering may enforce virtual simplicity, by highlighting the essential features and by separating them from the nice-to-have features.

Usability Engineering – UE

Usability engineering is the discipline for assuring the system's usability. Usability engineering implements human factors throughout the various disciplines involved in system engineering, to ensure that the system operation is fluent, efficient, reliable and safe. It is a cost-effective, user-centered process that ensures a high level of effectiveness, efficiency, and safety in complex interactive systems. Usability engineering is a structured, iterative, stepwise development process. Like the related disciplines of software and systems engineering, usability engineering is a combination of management principals and techniques, formal and semiformal evaluation techniques, and computerized tools.
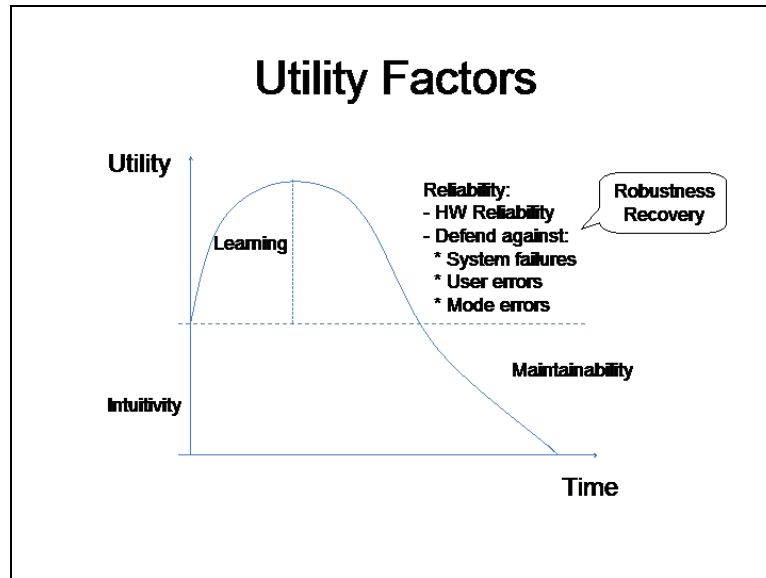
Definition of system usability. Usability is defined in many different ways, most of them emphasizing ease of use. The Usability Professional Association (UPA) defines usability as the degree to which something - software, hardware or anything else - is easy to use and a good fit for the people who use it (http://upassoc.org/usability_resources/about_usability/index.html) .

The examples above demonstrate that usability is much more than ease of use. ISO 9241-11 adds aspects of effectiveness and efficiency, defining usability as:

> " the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use."

However, our methodology is about another aspect of usability, namely, the human factors affecting the system utility.

Utility Assurance. Task-oriented SE enables maximizing the customer's utility in the long run. The utility function can be described as in the following chart:

The utility function has two phases: The startup and the main phase. The startup phase begins with the initial usage of the system and ends with the utility function reaching maximum utility. The initial value of the utility function is determined by the intuitivity of the user interface. The slope from the initial value to the maximum value is determined by the ease of learning. Following the startup phase is the beginning of the main phase, in which the utility function stabilizes. Then the system utility gradually decreases. The reasons for utility decrease include hardware reliability and maintenance costs, well know in common QA. Additional reasons for the decrease of the system utility may be attributed to human factors, such as user errors and the user's capability to handle system failures.

Task-oriented System Engineering

Traditionally, usability engineering focuses on the system intuitivity and ease of learning, which are features of the startup phase. Eventually, common usability practices are adequate to deal with these aspects and are of low value when dealing with the main phase of the utility function. Common QA practices on the other hand, are applicable to the main phase. However, they focus on technical aspects of the system, disregarding the user's role. The human factors that affect the main phase of the system utility are not considered by any of the common practices. This is why and where we need to extend the system engineering.

Sources of User Difficulties. The Task-oriented SE considers two sources of user difficulties:
- User errors
- User incapability to handle system failures.

An example of an accident due to a user error is the ecological disaster of 1967 caused by the Torrey Canyon supertanker (http://en.wikipedia.org/wiki/Torrey_Canyon). The accident was due to a combination of several exceptional events, the result of which was that the supertanker was heading directly to the rocks. At that point, the captain failed to change the course because the steering control lever was inadvertently set to the Control position, which disconnected the rudder from the wheel at the helm (Casey, 1998).

Examples of the second type are the TMI accident described above, the NYC blackout following a storm (http://en.wikipedia.org/wiki/New_York_City_blackout_of_1977) and the chemical plant disaster in Bhopal, India (http://en.wikipedia.org/wiki/Bhopal_Disaster).

Iterative design. Task-oriented SE is based on iterative design. In Technology-driven SE, the iterations enable changes in the specifications and design during the system testing. In Task-oriented SE, they enable early changes through prototyping and late changes following usability testing. The details of integrating usability engineering in SE are presented in Zonnenshein & Harel (2008).

Conclusion

Task-oriented SE enables us to make sure that the users not only use the system according to the specification, but also according to the customer's expectation. In particular, the Task-oriented SE approach presented here enables us to avoid user confusion and to defend the system from exceptional user events.

Bibliography

1. Buie, E., A., and Vallone, A. Integrating HCI engineering with software engineering: A call to a larger vision. In Smith, M. J., Salvendy, G., & Koubek, R. J. (Eds.), *Design of Computing Systems: Social and Ergonomic Considerations* (Proceedings of the Seventh International Conference on Human-Computer Interaction), Volume 2. Amsterdam, the Netherlands: Elsevier Science Publishers, 1997, pp. 525-530.
2. Case, S. E. Towards user-centered software engineering. *Proceedings of Usability Engineering 2: Measurement and Methods (UE2)*. Gaithersburg, MD, March, 1997, tbd pages.
3. Casey, S. *"Set Phasers on Stun"*, Aegean Publishing: Santa Barbara, 1998
4. Dorfman, M., S. *Introduction to Risk Management and Insurance (9th Edition)*. Englewood Cliffs, N.J: Prentice Hall. ISBN 0-13-224227-3. 2007
5. Dumas, J.S. and Redish, J.C., *"A Practical Guide to Usability Testing"*, Exeter, England; Portland, Or.: Intellect Books, 1999
6. Landawer, T.K., *"The Trouble with Computers: Usefulness, Usability, and Productivity"*, MIT Press, 1993
7. Leventhal, L., and Barnes J., *Usability Engineering, Process, Products & Examples*, Pearson Education, Inc., Pearson Prentice Hall, 2008.
8. Oliver, D. W., Kelliher, T.P., and Keegan, J.G., Engineering Complex Systems with Models and Objects. McGraw-Hill, New York, 1997
9. Paech, B., and Kohler, K., "Usability Engineering integrated with Requirements Engineering" ICSE Workshop *"Bridging the Gap between Software Engineering and Human-Computer Interaction"* 2003
10. Zonnenshein, A. & Harel, A, "Extended System Engineering – ESE: Integrating Usability Engineering in System Engineering". Poster presented at Incose International Symposium, Utrecht, The Netherlands, 2008. http://www.ergolight-sw.com/CHI/Company/Articles/ESE-Incose2008-P192.pdf

Appendix: The Usability Gap

The following table summarizes usability considerations typically missing from SE disciplines:

	Technology-driven Practices	**Typical Logical Gap**
System analysis	Excessive features satisfying marketing demands.	Users are slow. They fail to find the feature they need in time
System specification	Using SysML features	Popular SysML features hamper usability assurance:
Event-response definition	By use-cases	Enables user errors resulting in system failure: • Events that are unexpected and unacceptable in certain system states • System states that do not match the operational procedures
State definition	By state charts	Encourage mode-dependent system behavior, which enables mode errors
System architecture	Limited set of master requests	• Unlimited opportunities for user errors. • The system fails due to user errors
Interaction analysis	This is an informal activity. Operational procedures remain undefined. Users are expected to know the rules, although these are not defined yet.	Users unable to find the features they need, they do not know which option to select and what values to set.
Interaction specification and design	Informal based on use-cases, or by rapid prototyping by software experts	• Users do not follow the developer's intentions • Implements the error-prone mode-dependency
UI design	Attractors, such as animation, based on availability.	• Users distracted from their intentions • Users struggle to get their needs
Risk analysis	We focus on preventing system failures.	The system is not protected well against user errors. Mainly, the problem is that users fail to follow the system modes. (http://en.wikipedia.org/wiki/Mode_error)
System failure protection	We protect against expected failures.	• Users do not perceive the failure situation • Users do not recognize the system state • Users do not know how to resolve the problem
Error prevention	We assume users operate according to our intentions or instructions. We assume that users do not make mistakes and do not err	• Users model of the system is different from ours • Users make errors (Martin's cartoon: http://www.nevtron.si/borderline/delete.gif), often resulting in system failure.
User failure protection	We protect from risky events and from risky states	User events are sometimes unexpected, resulting in unexpected risky system states.
Testing	We assume that the users follow the (often undocumented) operational instructions	• Users operate not as presumed • The system does not handle unexpected user events • Critically risky unexpected user events not identified