Software Life cycle models

In the development of a software product, we consider two main parties: the customer who requires the use of the product and the designer who must provide the product. Typically, the customer and the designer are groups of people and some people can be both customer and designer. It is often important to distinguish between the customer who is the client of the designing company and the customer who is the eventual user of the system. These two roles of customer can be played by different people. The group of people who negotiate the features of the intended system with the designer may never be actual users of the system. This is often particularly true of web applications. In this chapter, we will use the term customer to refer to the group of people who interact with the design team and we will refer to those who will interact with the designed system as the user or end-user.

The graphical representation is reminiscent of a waterfall, in which each activity naturally leads into the next. The analogy of the waterfall is not completely faithful to the real relationship between these activities, but it provides a good starting point for discussing the logical flow of activity. We describe the activities of this waterfall model of the software life cycle .

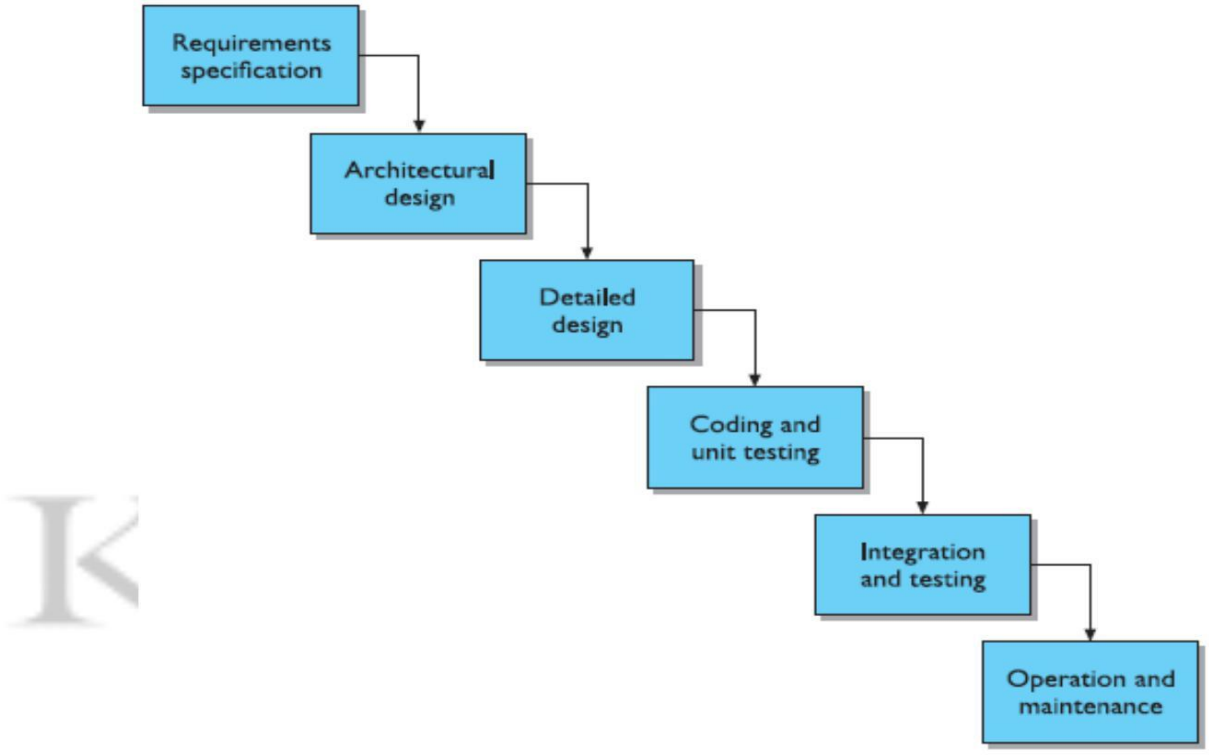

Figure The activities in the waterfall model of the software life cycle

Requirements specification

Requirements specification begins at the start of product development. Though the requirements are from the customer‘s perspective, if they are to be met by the software product they must be formulated in a language suitable for implementation. Requirements are usually initially expressed in the native language of the customer.

The executable languages for software are less natural and are more closely related to a mathematical language in which each term in the language has a precise interpretation, or semantics. The transformation from the expressive but relatively ambiguous natural language of requirements to the more precise but less expressive executable languages is one key to successful development. Task analysis techniques, which are used to express work domain requirements in a form that is both expressive and precise.

Architectural design

The requirements specification concentrates on what the system is supposed to do. The next activities concentrate on how the system provides the services expected from it. The first activity is a high-level decomposition of the system into components that can either be brought in from existing software products or be developed from scratch independently. An architectural design performs this decomposition. It is not only concerned with the functional decomposition of the system, determining which components provide which services. It must also describe the interdependencies between separate components and the sharing of resources that will arise between components.

Detailed design

The architectural design provides a decomposition of the system description that allows for isolated development of separate components which will later be integrated. For those components that are not already available for immediate integration, the designer must provide a sufficiently detailed description so that they may be implemented in some programming language. The detailed design is a refinement of the component description provided by the architectural design. The behaviour implied by the higher-level description must be preserved in the more detailed description.

There will be more than one possible refinement of the architectural component that will satisfy the behavioural constraints. Choosing the best refinement is often a matter of trying to satisfy as

many of the non-functional requirements of the system as possible. Thus the language used for the detailed design must allow some analysis of the design in order to assess its properties.

Coding and unit testing

The detailed design for a component of the system should be in such a form that it is possible to implement it in some executable programming language. After coding, the component can be tested to verify that it performs correctly, according to some test criteria that were determined in earlier activities. Research on this activity within the life cycle has concentrated on two areas. There is plenty of research that is geared towards the automation of this coding activity directly from a low-level detailed design. Most of the work in formal methods operates under the hypothesis that, in theory, the transformation from the detailed design to the implementation is from one mathematical representation to another and so should be able to be entirely automated. Other, more practical work concentrates on the automatic generation of tests from output of earlier activities which can be performed on a piece of code to verify that it behaves correctly.

Integration and testing

Once enough components have been implemented and individually tested, they must be integrated as described in the architectural design. Further testing is done to ensure correct behaviour and acceptable use of any shared resources. It is also possible at this time to perform some acceptance testing with the customers to ensure that the system meets their requirements. It is only after acceptance of the integrated system that the product is finally released to the customer.

Maintenance

After product release, all work on the system is considered under the category of maintenance, until such time as a new version of the product demands a total redesign or the product is phased out entirely. Consequently, the majority of the lifetime of a product is spent in the maintenance activity. Maintenance involves the correction of errors in the system which are discovered after release and the revision of the system services to satisfy requirements that were not realized during previous development.

Validation and verification

Throughout the life cycle, the design must be checked to ensure that it both satisfies the high-level requirements agreed with the customer and is also complete and internally consistent. These checks are referred to as validation and verification, respectively. Verification of a design

will most often occur within a single life-cycle activity or between two adjacent activities. For example, in the detailed design of a component of a payroll accounting system, the designer will be concerned with the correctness of the algorithm to compute taxes deducted from an employee‘s gross income.

The architectural design will have provided a general specification of the information input to this component and the information it should output. The detailed description will introduce more information in refining the general specification. The detailed design may also have to change the representations for the information and will almost certainly break up a single high-level operation into several low-level operations that can eventually be implemented. In introducing these changes to information and operations, the designer must show that the refined description is a legal one within its language (internal consistency) and that it describes all of the specified behaviour of the high-level description (completeness) in a provably correct way (relative consistency). Validation of a design demonstrates that within the various activities the customer‘s requirements are satisfied.

Validation is a much more subjective exercise than verification, mainly because the disparity between the language of the requirements and the language of the design forbids any objective form of proof. In interactive system design, the validation against HCI requirements is often referred to as evaluation and can be performed by the designer in isolation or in cooperation with the customer.

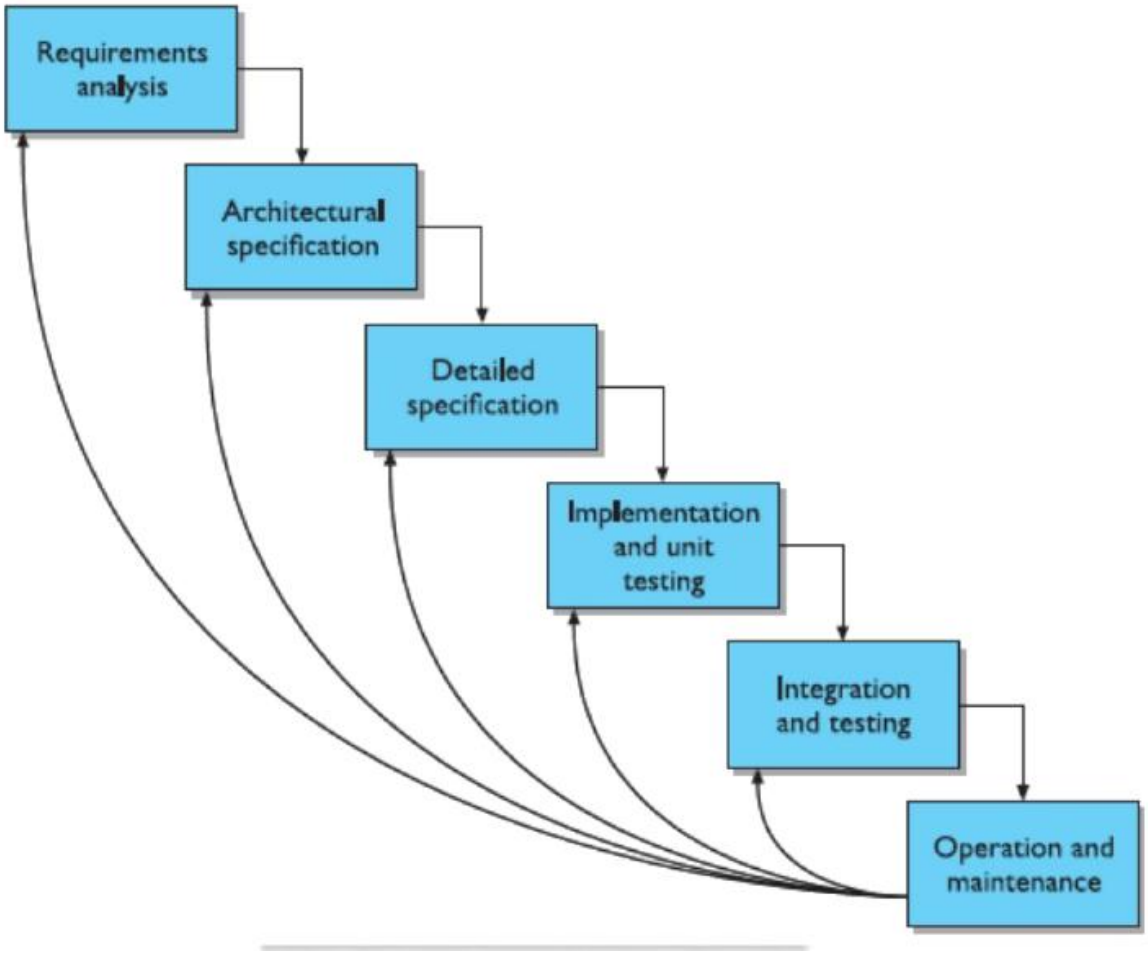

Figure: Feedback from maintenance activity to other design activities

Management and contractual issues

The life cycle described above concentrated on the more technical features of software development. In a technical discussion, managerial issues of design, such as time constraints and economic forces, are not as important. The different activities of the life cycle are logically related to each other. We can see that requirements for a system precede the high-level architectural design which precedes the detailed design, and so on. In reality, it is quite possible that some detailed design is attempted before all of the architectural design. In management, a much wider perspective must be adopted which takes into account the marketability of a system, its training needs, the availability of skilled personnel or possible subcontractors, and other topics outside the activities for the development of the isolated system.

Interactive systems and the software life cycle

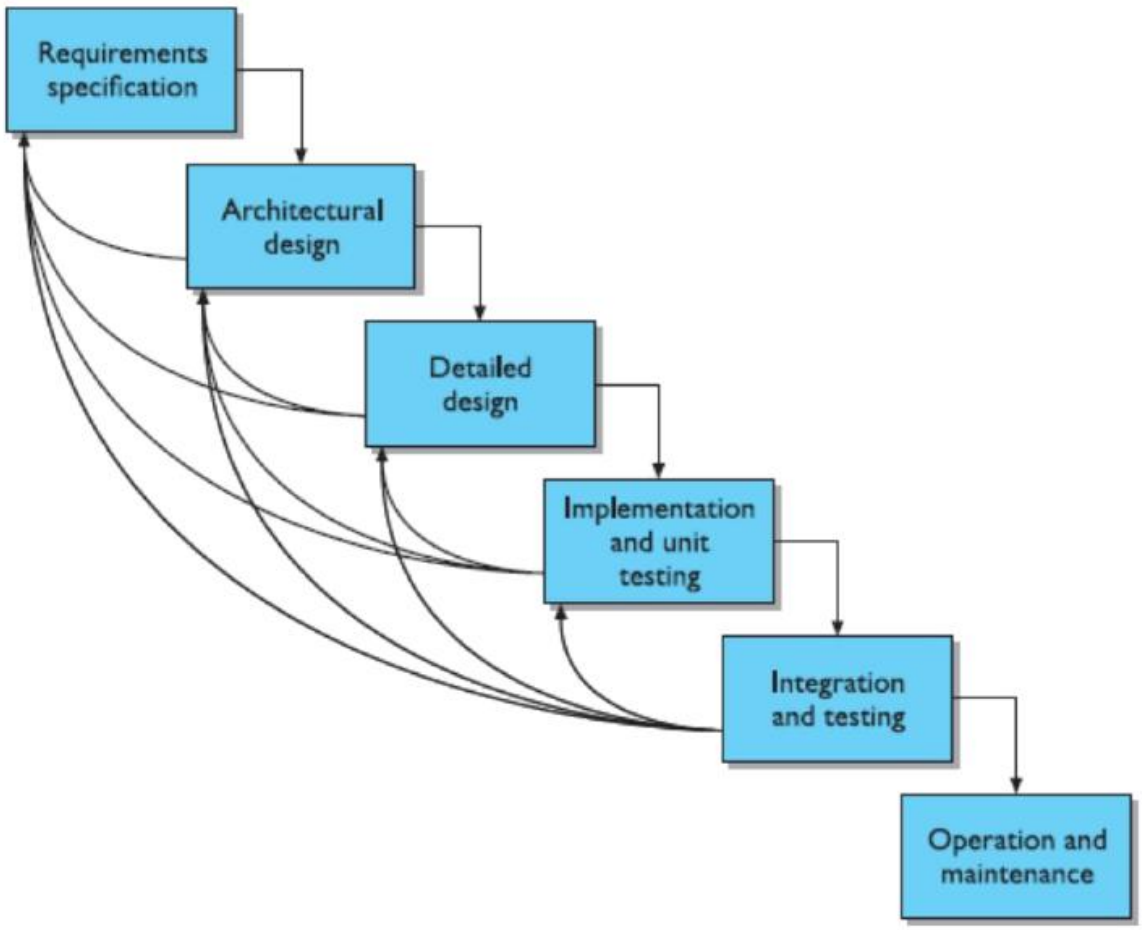

Figure: Representing iteration in the waterfall model

The life cycle for development we described above presents the process of design in a somewhat pipeline order. In reality, even for batch-processing systems, the actual design process is iterative, work in one design activity affecting work in any other activity either before or after it in the life cycle. A final point about the traditional software life cycle is that it does not promote the use of notations and techniques that support the user‘s perspective of the interactive system. We discussed earlier the purpose of validation and the formality gap.

It is very difficult for an expert on human cognition to predict the cognitive demands that an abstract design would require of the intended user if the notation for the design does not reflect the kind of information the user must recall in order to interact. The same holds for assessing the timing behaviour of an abstract design that does not explicitly mention the timing characteristics of the operations to be invoked or their relative ordering.

Though no structured development process will entirely eliminate the formality gap, the particular notations used can go a long way towards making validation of non-functional requirements feasible with expert assistance. In the remaining sections of this chapter, we will describe various approaches to augment the design process to suit better the design of interactive systems. These approaches are categorized under the banner of user-centred design.

USABILITY ENGINEERING

In relation to the software life cycle, one of the important features of usability engineering is the inclusion of a usability specification, forming part of the requirements specification that concentrates on features of the user–system interaction which contribute to the usability of the product. Various attributes of the system are suggested as gauges for testing the usability. For each attribute, six items are defined to form the usability specification of that attribute**.**

Table : Sample usability specification for undo with a VCR

Attribute:	Backward recoverability
Measuring concept:	Undo an erroneous programming sequence
Measuring method:	Number of explicit user actions to undo current program
Now level:	No current product allows such an undo
Worst case:	As many actions as it takes to program in mistake
Planned level:	A maximum of two explicit user actions
Best case:	One explicit cancel action

Recoverability refers to the ability to reach a desired goal after recognition of some error in previous interaction. The recovery procedure can be in either a backward or forward sense. Current VCR design has resulted in interactive systems that are notoriously difficult to use; the redesign of a VCR provides a good case study for usability engineering. In designing a new VCR control panel, the designer wants to take into account how a user might recover from a mistake he discovers while trying to program the VCR to record some television program in his absence. One approach that the designer decides to follow is to allow the user the ability to undo the programming sequence, reverting the state of the VCR to what it was before the programming task began.

The backward recoverability attribute is defined in terms of a measuring concept, which makes the abstract attribute more concrete by describing it in terms of the actual product. So in this

case, we realize backward recoverability as the ability to undo an erroneous programming sequence. The measuring method states how the attribute will be measured, in this case by the number of explicit user actions required to perform the undo, regardless of where the user is in the programming sequence. The remaining four entries in the usability specification then provide the agreed criteria for judging the success of the product based on the measuring method. The now level indicates the value for the measurement with the existing system, whether it is computer based or not.

The worst case value is the lowest acceptable measurement for the task, providing a clear distinction between what will be acceptable and what will be unacceptable in the final product. The planned level is the target for the design and the best case is the level which is agreed to be the best possible measurement given the current state of development tools and technology. In the example, the designers can look at their previous VCR products and those of their competitors to determine a suitable now level. In this case, it is determined that no current model allows an undo which returns the state of the VCR to what it was before the programming task.

Table: Criteria by which measuring method can be determined

1. Time to complete a task
2. Per cent of task completed
3. Per cent of task completed per unit time
4. Ratio of successes to failures
5. Time spent in errors
6. Per cent or number of errors
7. Per cent or number of competitors better than it
8. Number of commands used
9. Frequency of help and documentation use
10. Per cent of favorable/unfavorable user comments
11. Number of repetitions of failed commands
12. Number of runs of successes and of failures
13. Number of times interface misleads the user
14. Number of good and bad features recalled by users
15. Number of available commands not invoked
16. Number of regressive behaviors

17. Number of users preferring your system
18. Number of times users need to work around a problem
19. Number of times the user is disrupted from a work task
20. Number of times user loses control of the system
21. Number of times user expresses frustration or satisfaction

Table : Possible ways to set measurement levels in a usability specification

Set levels with respect to information on:
1. an existing system or previous version
2. competitive systems
3. carrying out the task without use of a computer system
4. an absolute scale
5. your own prototype
6. user's own earlier performance
7. each component of a system separately
8. a successive split of the difference between best and worst values observed in user tests

Table: Examples of usability metrics from ISO 9241

Usability objective	Effectiveness measures	Efficiency measures	Satisfaction measures
Suitability for the task	Percentage of goals achieved	Time to complete a task	Rating scale for satisfaction
Appropriate for trained users	Number of power features used	Relative efficiency compared with an expert user	Rating scale for satisfaction with power features
Learnability	Percentage of functions learned	Time to learn criterion	Rating scale for ease of learning
Error tolerance	Percentage of errors corrected successfully	Time spent on correcting errors	Rating scale for error handling

Problems with usability engineering

The major feature of usability engineering is the assertion of explicit usability metrics early on in the design process which can be used to judge a system once it is delivered. There is a very solid argument which points out that it is only through empirical approaches such as the use of

usability metrics that we can reliably build more usable systems. Although the ultimate yardstick for determining usability may be by observing and measuring user performance, that does not mean that these measurements are the best way to produce a predictive design process for usability.

The problem with usability metrics is that they rely on measurements of very specific user actions in very specific situations. When the designer knows what the actions and situation will be, then she can set goals for measured observations. However, at early stages of design, designers do not have this information. Take our example usability specification for the VCR. In setting the acceptable and unacceptable levels for backward recovery, there is an assumption that a button will be available to invoke the undo. In fact, the designer was already making an implicit assumption that the user would be making errors in the programming of the VCR. We should recognize another inherent limitation for usability engineering, which provides a means of satisfying usability specifications and not necessarily usability. The designer is still forced to understand why a particular usability metric enhances usability for real people.

ITERATIVE DESIGN AND PROTOTYPING

The design can then be modified to correct any false assumptions that were revealed in the testing. This is the essence of iterative design, a purposeful design process which tries to overcome the inherent problems of incomplete requirements specification by cycling through several designs, incrementally improving upon the final product with each pass.

The problems with the design process, which lead to an iterative design philosophy, are not unique to the usability features of the intended system. The problem holds for requirements specification in general, and so it is a general software engineering problem, together with technical and managerial issues. On the technical side, iterative design is described by the use of prototypes, artifacts that simulate or animate some but not all features of the intended system.

There are three main approaches to prototyping:

Throw-away :The prototype is built and tested. The design knowledge gained from this exercise is used to build the final product, but the actual prototype is discarded. Figure depicts the procedure in using throw-away prototypes to arrive at a final requirements specification in order for the rest of the design process to proceed.

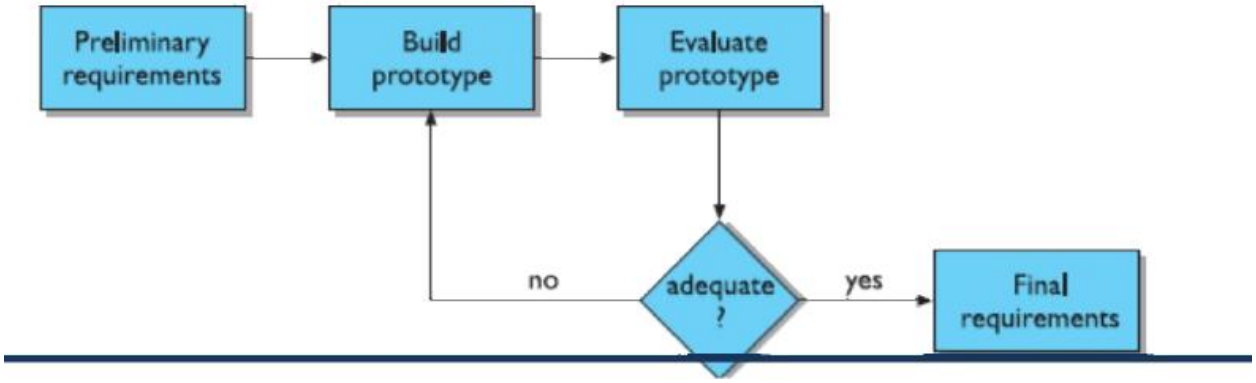

Figure: Throw-away prototyping within requirements specification

Incremental The final product is built as separate components, one at a time. There is one overall design for the final system, but it is partitioned into independent and smaller components. The final product is then released as a series of products, each subsequent release including one more component.

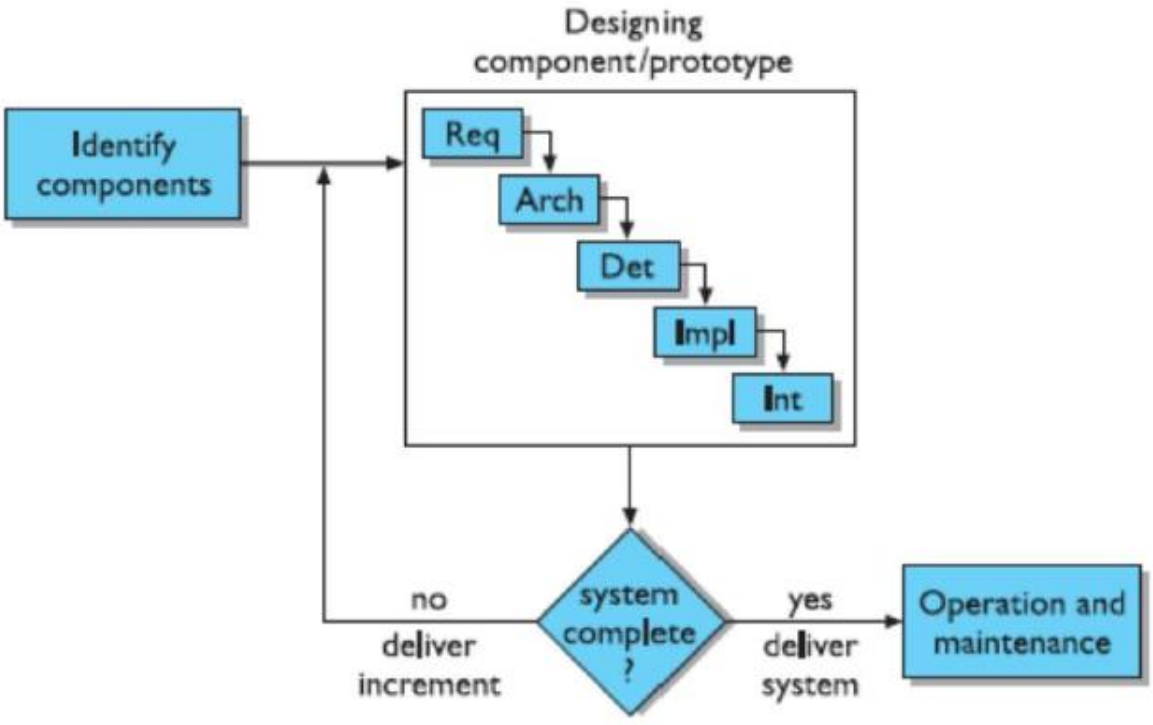

Figure: Incremental prototyping within the life cycle

Evolutionary Here the prototype is not discarded and serves as the basis for the next iteration of design. In this case, the actual system is seen as evolving from a very limited initial

version to its final release, Evolutionary prototyping also fits in well with the modifications which must be made to the system that arise during the operation and maintenance activity in the life cycle.

Prototypes differ according to the amount of functionality and performance they provide relative to the final product. An animation of requirements can involve no real functionality, or limited functionality to simulate only a small aspect of the interactive behavior for evaluative purposes. At the other extreme, full functionality can be provided at the expense of other performance characteristics, such as speed or error tolerance.

Regardless of the level of functionality, the importance of a prototype lies in its projected realism. The prototype of an interactive system is used to test requirements by evaluating their impact with real users. An honest appraisal of the requirements of the final system can only be trusted if the evaluation conditions are similar to those anticipated for the actual operation. But providing realism is costly, so there must be support.

Time Building prototypes takes time and, if it is a throw-away prototype, it can be seen as precious time taken away from the real design task. So the value of prototyping is only appreciated if it is fast, hence the use of the term rapid prototyping. Rapid development and manipulation of a prototype should not be mistaken for rushed evaluation which might lead to erroneous results and invalidate the only advantage of using a prototype in the first place.

Planning Most project managers do not have the experience necessary for adequately planning and costing a design process which involves prototyping. Non-functional features Often the most important features of a system will be nonfunctional ones, such as safety and reliability, and these are precisely the kinds of features which are sacrificed in developing a prototype. For evaluating usability features of a prototype, response time – yet another feature often compromised in a prototype – could be critical to product acceptance. This problem is similar to the technical issue of prototype realism.

Contracts The design process is often governed by contractual agreements between customer and designer which are affected by many of these managerial and technical issues. Prototypes and other implementations cannot form the basis for a legal contract, and so an iterative design process will still require documentation which serves as the binding agreement. There must be an effective way of translating the results derived from prototyping into adequate documentation. A

rapid prototyping process might be amenable to quick changes, but that does not also apply to the design process.

Techniques for prototyping

Probably the simplest notion of a prototype is the storyboard, which is a graphical depiction of the outward appearance of the intended system, without any accompanying system functionality. Storyboards do not require much in terms of computing power to construct; in fact, they can be mocked up without the aid of any computing resource. The origins of storyboards are in the film industry, where a series of panels roughly depicts snapshots from an intended film sequence in order to get the idea across about the eventual scene. Similarly, for interactive system design, the storyboards provide snapshots of the interface at particular points in the interaction. Evaluating customer or user impressions of the storyboards can determine relatively quickly if the design is heading in the right direction. Modern graphical drawing packages now make it possible to create storyboards with the aid of a computer instead of by hand. Though the graphic design achievable on screen may not be as sophisticated as that possible by a professional graphic designer, it is more realistic because the final system will have to be displayed on a screen. Also, it is possible to provide crude but effective animation by automated sequencing through a series of snapshots. Animation illustrates the dynamic aspects of the intended user–system interaction, which may not be possible with traditional paper-based storyboards. If not animated, storyboards usually include annotations and scripts indicating how the interaction will occur.

Limited functionality simulations

Storyboards and animation techniques are not sufficient for this purpose, as they cannot portray adequately the interactive aspects of the system. To do this, some portion of the functionality must be simulated by the design team. Programming support for simulations means a designer can rapidly build graphical and textual interaction objects and attach some behaviour to those objects, which mimics the system‘s functionality.

Once this simulation is built, it can be evaluated and changed rapidly to reflect the results of the evaluation study with various users. High-level programming support HyperTalk and many similar languages allow the programmer to attach functional behavior to the specific interactions that the user will be able to do, such as position and click on the mouse over a button on the screen. Previously, the difficulty of interactive programming was that it was so implementation

dependent that the programmer would have to know quite a bit of intimate detail of the hardware system in order to control even the simplest of interactive behavior. These high-level programming languages allow the programmer to abstract away from the hardware specifics and think in terms that are closer to the way the input and output devices are perceived as interaction devices. The frequent conceptual model put forth for interactive system design is to separate the application functionality from its presentation. It is then possible to program the underlying functionality of the system and to program the behavior of the user interface separately. The job of a UIMS, then, is to allow the programmer to connect the behavior at the interface with the underlying functionality.

Warning about iterative design

The ideal model of iterative design, in which a rapid prototype is designed, evaluated and modified until the best possible design is achieved in the given project time, is appealing. But there are two problems. First, it is often the case that design decisions made at the very beginning of the prototyping process are wrong and, in practice, design inertia can be so great as never to overcome an initial bad decision. So, whereas iterative design is, in theory, amenable to great changes through iterations, it can be the case that the initial prototype has bad features that will not be amended.

The second problem is slightly more subtle, and serious. If, in the process of evaluation, a potential usability problem is diagnosed, it is important to understand the reason for the problem and not just detect the symptom.

DESIGN RATIONALE

Design rationale is the information that explains why a computer system is the way it is, including its structural or architectural description and its functional or behavioural description. In this sense, design rationale does not fit squarely into the software life cycle described in this chapter as just another phase or box. Rather, design rationale relates to an activity of both reflection (doing design rationale) and documentation (creating a design rationale) that occurs throughout the entire life cycle.

In an explicit form, a design rationale provides a communication mechanism among the members of a design team so that during later stages of design and/or maintenance it is possible to understand what critical decisions were made, what alternatives were investigated (and, possibly, in what order) and the reason why one alternative was chosen over the others.

This can help avoid incorrect assumptions later.

 Accumulated knowledge in the form of design rationales for a set of products can bereused to transfer what has worked in one situation to another situation which has similar needs. The design rationale can capture the context of a design decision in order that a different design team can determine if a similar rationale is appropriate for their product.

 The effort required to produce a design rationale forces the designer to deliberate more carefully about design decisions. The process of deliberation can be assisted by the design rationale technique by suggesting how arguments justifying or discarding a particular design option are formed.

In the area of HCI, design rationale has been particularly important, again for several reasons:

 There is usually no single best design alternative. More often, the designer is faced with a set of trade-offs between different alternatives. For example, a graphical interface may involve a set of actions that the user can invoke by use of the mouse and the designer must decide whether to present each action as a _button‘ on the screen, which is always visible, or hide all of the actions in a menu which must be explicitly invoked before an action can be chosen. The former option maximizes the operation visibility but the latter option takes up less screen space. It would be up to the designer to determine which criterion for evaluating the options was more Important and then communicating that information in a design rationale.

 Even if an optimal solution did exist fora given design decision, the space of alternatives is so vast that it is unlikely a designer would discover it. In this case, it is important that the designer indicates all alternatives that have been investigated. Then later on it can be determined if she has not considered the best solution or had thought about it and discarded it for some reason. In project management, this kind of accountability for design is good.

 The usability of an interactive system is very dependent on the context of its use. The flashiest graphical interface is of no use if the end-user does not have access to a highquality graphics display or a pointing device. Capturing the context in which a design decision is made will help later when new products are designed.If the context remains the same, then the old rationale can be adopted without revision. If the context has changed somehow, the old rationale can be re-examined to see if any rejected alternatives are now more favourable or if any new alternatives are now possible. Process-oriented design rationale Rationale is based on Rittel‘s issue-based information system, or IBIS, a style for representing design and planning dialog developed in the

1970s. In IBIS (pronounced ‗ibbiss‘), a hierarchical structure to a design rationale is created. A root issue is identified which represents the main problem or question that the argument is addressing. Various positions are put forth as potential resolutions for the root issue, and these are depicted as descendants in the IBIS hierarchy directly connected to the root issue. Each position is then supported or refuted by arguments, which modify the relationship between issue and position.

The hierarchy grows as secondary issues are raised which modify the root issue in some way.

Each of these secondary issues is in turn expanded by positions and arguments, further subissues, and so on.

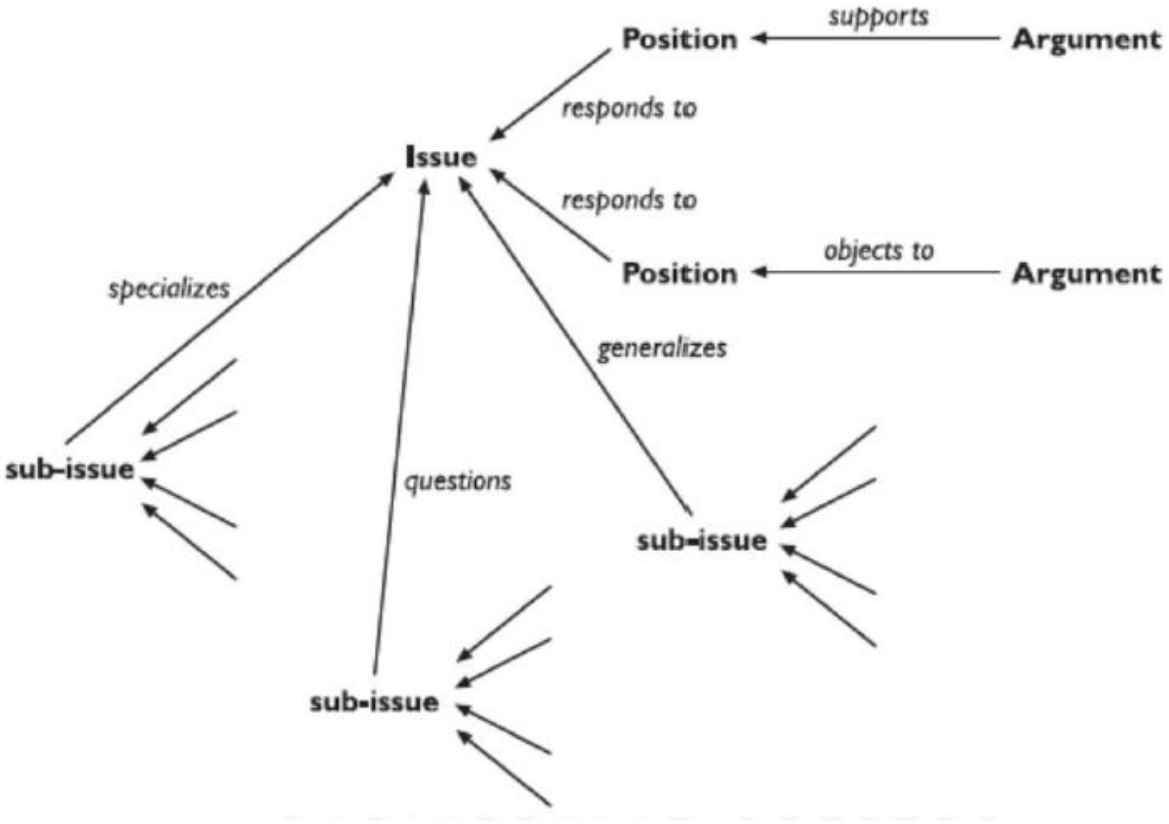

Figure: The structure of a gIBIS design rationale

A graphical version of IBIS has been defined by Conklin and Yakemovic called gIBIS (pronounced ‗gibbiss‘), which makes the structure of the design rationale more apparent visually in the form of a directed graph which can be directly edited by the creator of the design rationale. Above figure gives a representation of the gIBIS vocabulary. Issues, positions and arguments are nodes in the graph and the connections between them are labeled to clarify the relationship

between adjacent nodes. So, for example, an issue can suggest further sub-issues, or a position can respond to an issue or an argument can support a position. The gIBIS structure can be supported by a hypertext tool to allow a designer to create and browse various parts of the design rationale.

Design space analysis

MacLean and colleagues have proposed a more deliberative approach to design rationale which emphasizes a post hoc structuring of the space of design alternatives that have been considered in a design project. Their approach, embodied in the Questions, Options and Criteria (QOC) notation, is characterized as design space analysis issues raised based on reflection and understanding of the actual design activities. Questions in a design space analysis are therefore similar to issues in IBIS except in the way they are captured. Options provide alternative solutions to the question. They are assessed according to some criteria in order to determine the most favorable option. In Figure an option which is favorably assessed in terms of a criterion is linked with a solid line, whereas negative links have a dashed line.

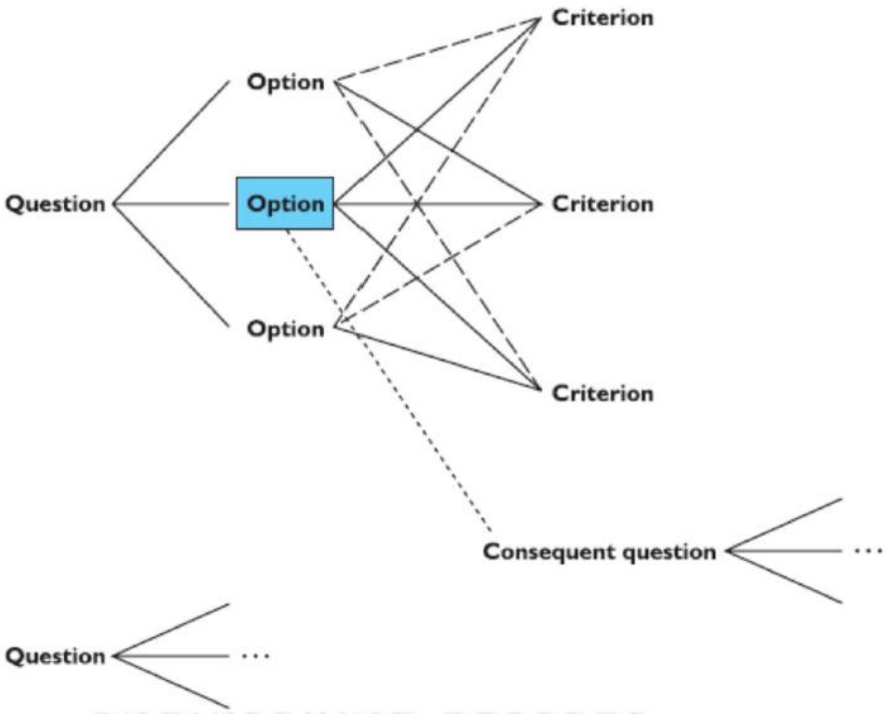

Figure: The QOC notation

The key to an effective design space analysis using the QOC notation is deciding the right questions to use to structure the space and the correct criteria to judge the options. The initial questions raised must be sufficiently general that they cover a large enough portion of the possible design space, but specific enough that a range of options can be clearly identified. It can be difficult to decide the right set of criteria with which to assess the options.

Structure-oriented technique, called Decision Representation Language (DRL), developed by Lee and Lai, structures the design space in a similar fashion to QOC, though its language is somewhat larger and it has a formal semantics. The questions, options and criteria in DRL are given the names: decision problem, alternatives and goals. QOC assessments are represented in DRL by a more complex language for relating goals to alternatives. The sparse language in QOC used to assess an option relative to a criterion (positive or negative assessment only) is probably insufficient, but there is a trade-off involved in adopting a more complex vocabulary which may prove too difficult to use in practice. The advantage of the formal semantics of DRL is that the design rationale can be used as a computational mechanism to help manage the large volume of information. For example, DRL can track the dependencies between different decision problems, so that subsequent changes to the design rationale for one decision problem can be automatically propagated to other dependent problems. Design space analysis directly addresses the claim that no design activity can hope to uncover all design possibilities, so the best we can hope to achieve is to document the small part of the design space that has been investigated. An advantage of the post hoc technique is that it can abstract away from the particulars of a design meeting and therefore represent the design knowledge in such a way that it can be of use in the design of other products. The major disadvantage is the increased overhead such an analysis warrants. More time must be taken away from the design activity to do this separate documentation task. When time is scarce, these kinds of overhead costs are the first to be trimmed.

Psychological design rationale

The final category of design rationale tries to make explicit the psychological claims of usability inherent in any interactive system in order better to suit a product for the tasks users have. This psychological design rationale has been introduced by Carroll and Rosson, and before we describe the application of the technique it is important to understand some of its theoretical background.

When designing a new interactive system, the designers take into account the tasks that users currently perform and any new ones that they may want to perform. This task identification serves as part of the requirements for the new system, and can be done through empirical observation of how people perform their work currently and presented through informal language or a more formal task analysis language . When the new system is implemented, or becomes an artifact, further observation reveals that in addition to the required tasks it was built to support, it also supports users in tasks that the designer never intended. Once designers understand these new tasks, and the associated problems that arise between them and the previously known tasks, the new task definitions can serve as requirements for future artifacts.

Carroll refers to this real-life phenomenon as the task–artifact cycle. He provides a good example of this cycle through the evolution of the electronic spreadsheet. When the first electronic spreadsheet, VisiCalc, was marketed in the late 1970s, it was presented simply as an automated means of supporting tabular calculation, a task commonly used in the accounting world.

Within little over a decade of its introduction, the application of spreadsheets had far outstripped its original intent within accounting. Spreadsheets were being used for all kinds of financial analysis, ‗what-if ' simulations, report formatting and even as a general programming language paradigm! As the set of tasks expands, new spreadsheet products have flooded the marketplace trying to satisfy the growing customer base. Another good example of the task–artifact cycle in action is with word processing, which was originally introduced to provide more automated support for tasks previously achieved with a typewriter and now provides users with the ability to carry out various authoring tasks that they never dreamed possible with a conventional typewriter. And today the tasks for the spreadsheet and the word processor are intermingled in the same artifact.

The purpose of psychological design rationale is to support this natural task– artifact cycle of design activity. The main emphasis is not to capture the designer's intention in building the artifact. Rather, psychological design rationale aims to make explicit the consequences of a design for the user, given an understanding of what tasks he intends to perform. Previously, these psychological consequences were left implicit in the design, though designers would make informal claims about their systems

The first step in the psychological design rationale is to identify the tasks that the proposed system will address and to characterize those tasks by questions that the user tries to answer

in accomplishing them. For instance, Carroll gives an example of designing a system to help programmers learn the Smalltalk object-oriented programming language environment. The main task the system is to support is learning how Smalltalk works. In learning about the programming environment, the programme will perform tasks that help her answer the questions:

- What can I do: that is, what are the possible operations or functions that this programming environment allows?
- How does it work: that is, what do the various functions do?
- How can I do this: that is, once I know a particular operation I want to perform,
- how do I go about programming it?

DESIGN RULES

- Designing for maximum usability is the goal of interactive systems design.
- Abstract principles offer a way of understanding usability in a more general sense, especially if we can express them within some coherent catalog.
- Design rules in the form of standards and guidelines provide direction for design, in both general and more concrete terms, in order to enhance the interactive properties of the system.
- The essential characteristics of good design are often summarized through ‗golden rules' or heuristics.
- Design patterns provide a potentially generative approach to capturing and reusing design knowledge.

PRINCIPLES TO SUPPORT USABILITY

The principles we present are first divided into three main categories:

Learnability – the ease with which new users can begin effective interaction and achieve maximal performance.

Flexibility – the multiplicity of ways in which the user and system exchange information.

Robustness – the level of support provided to the user in determining successful achievement and assessment of goals.

Table :Summary of principles affecting learnability

Principle	Definition	Related principles
Predictability	Support for the user to determine the effect of future action based on past interaction history	Operation visibility
Synthesizability	Support for the user to assess the effect of past operations on the current state	Immediate/eventual honesty
Familiarity	The extent to which a user's knowledge and experience in other real-world or computer-based domains can be applied when interacting with a new system	Guessability, affordance
Generalizability	Support for the user to extend knowledge of specific interaction within and across applications to other similar situations	–
Consistency	Likeness in input–output behavior arising from similar situations or similar task objectives	–

Predictability

Predictability of an interactive system is distinguished from deterministic behavior of the computer system alone. Most computer systems are ultimately deterministic machines, so that given the state at any one point in time and the operation which is to be performed at that time, there is only one possible state that can result. Predictability is a user-centered concept; it is deterministic behavior from the perspective of the user. It is not enough for the behavior of the computer system to be determined completely from its state, as the user must be able to take advantage of the determinism.

Synthesizability

When an operation changes some aspect of the internal state, it is important that the change is seen by the user. The principle of honesty relates to the ability of the user interface to provide an observable and informative account of such change. In the best of circumstances, this notification can come immediately, requiring no further interaction initiated by the user. At the very least, the notification should appear eventually, after explicit user directives to make the change observable. A good example of the distinction between immediacy and eventuality can be seen in the comparison between command language interfaces and visual desktop interfaces for a file management system. You have moved a file from one directory to another. The principle of honesty implies that after moving the file to its new location in the file system you are then able to determine its new whereabouts. In a command language system, you would

typically have to remember the destination directory and then ask to see the contents of that directory in order to verify that the file has been moved (in fact, you would also have to check that the file is no longer in its original directory to determine that it has been moved and not copied). In a visual desktop interface, a visual representation (or icon) of the file is dragged from its original directory and placed in its destination directory where it remains visible (assuming the destination folder is selected to reveal its contents). In this case, the user need not expend any more effort to assess the result of the move operation.The visual desktop is immediately honest.

Familiarity

New users of a system bring with them a wealth of experience across a wide number of application domains. This experience is obtained both through interaction in thereal world and through interaction with other computer systems. For a new user, the familiarity of an interactive system measures the correlation between the user‘s existing knowledge and the knowledge required for effective interaction. For example, when word processors were originally introduced the analogy between the word processor and a typewriter was intended to make the new technology more immediately accessible to those who had little experience with the former but a lot of experience with the latter. Familiarity has to do with a user‘s first impression of the system. In this case, we are interested in how the system is first perceived and whether the user can determine how to initiate any interaction.

Generalizability

The generalizability of an interactive system supports this activity, leading to a more complete predictive model of the system for the user. We can apply generalization to situations in which the user wants to apply knowledge that helps achieve one particular goal to another situation where the goal is in some way similar. Generalizability can be seen as a form of consistency. Generalization can occur within a single application or across a variety of applications. For example, in a graphical drawing package that draws a circle as a constrained form of ellipse, we would want the user to generalize that a square can be drawn as a constrained rectangle. A good example of generalizability across a variety of applications

can be seen in multi-windowing systems that attempt to provide cut/paste/copy operations to all applications in the same way (with varying degrees of success). Generalizability within an application can be maximized by any conscientious designer.

Consistency

Consistency relates to the likeness in behavior arising from similar situations or similar task objectives. Consistency is probably the most widely mentioned principle in the literature on user interface design. _Be consistent!‘ we are constantly urged. The user relies on a consistent interface. However, the difficulty of dealing with consistency is that it can take many forms. Consistency is not a single property of an interactive system that is either satisfied or not satisfied. Instead, consistency must be applied relative to something. Thus we have consistency in command naming, or consistency in command/argument invocation.

Consistency can be expressed in terms of the form of input expressions or output responses with respect to the meaning of actions in some conceptual model of the system. For example, before the introduction of explicit arrow keys, some word processors used the relative position of keys on the keyboard to indicate directionality for operations (for example, to move one character to the left, right, up or down).The conceptual model for display-based editing is a two-dimensional plane, so the user would think of certain classes of operations in terms of movements up, down, left or right in the plane of the display. Operations that required directional information, such as moving within the text or deleting some unit of text, could be articulated by using some set of keys on the keyboard that form a pattern consistent with up, down, left and right (for example, the keys e, x, s and d, respectively). For output responses, a good example of consistency can be found in a warnings system for an aircraft. Warnings to the pilot are classified into three categories, depending on whether the situation with the aircraft requires immediate recovery action, eventual but not immediate action, or no action at all (advisory) on the part of the crew.

Flexibility

Principle	Definition	Related principles
Dialog initiative	Allowing the user freedom from artificial constraints on the input dialog imposed by the system	System/user pre-emptiveness
Multi-threading	Ability of the system to support user interaction pertaining to more than one task at a time	Concurrent vs. interleaving, multi-modality
Task migratability	The ability to pass control for the execution of a given task so that it becomes either internalized by the user or the system or shared between them	–
Substitutivity	Allowing equivalent values of input and output to be arbitrarily substituted for each other	Representation multiplicity, equal opportunity
Customizability	Modifiability of the user interface by the user or the system	Adaptivity, adaptability

Table: Summary of principles affecting flexibility Dialog

Dialog initiative

The system can initiate all dialog, in which case the user simply responds to requests for information. We call this type of dialog system pre-emptive. For example, a modal dialog box prohibits the user from interacting with the system in any way that does not direct input to the box. Alternatively, the user may be entirely free to initiate any action towards the system, in which case the dialog is user pre-emptive. The system may control the dialog to the extent that it prohibits the user from initiating any other desired communication concerning the current task or some other task the user would like to perform. From the user‘s perspective, a system-driven interaction hinders flexibility whereas a user-driven interaction favours it.

In general, we want to maximize the user‘s ability to pre-empt the system and minimize the system‘s ability to pre-empt the user. Although a system pre-emptive dialog is not desirable in general, some situations may require it. In a cooperative editor (in which two people edit a document at the same time) it would be impolite for you to erase a paragraph of text that your partner is currently editing. For safety reasons, it may be necessary to prohibit the user from the ‗freedom‘ to do potentially serious damage. A pilot about to land an aircraft in which the flaps have asymmetrically failed in their extended position2 should not be allowed to abort the landing, as this failure will almost certainly result in a catastrophic accident.

Multi-threading

A thread of a dialog is a coherent subset of that dialog. In the user–system dialog, we can consider a thread to be that part of the dialog that relates to a given user task. Multi-threading of the user–system dialog allows for interaction to support more than one task at a time. Concurrent multi-threading allows simultaneous communication of information pertaining to separate tasks. Interleaved multi-threading permits a temporal overlap between separate tasks, but stipulates that at any given instant the dialog is restricted to a single task.

Task migratability

Task migratability concerns the transfer of control for execution of tasks between system and user. It should be possible for the user or system to pass the control of a task over to the other or promote the task from a completely internalized one to a shared and cooperative venture. Hence, a task that is internal to one can become internal to the other or shared between the two partners.

Substitutivity

Substitutivity requires that equivalent values can be substituted for each other. For example, in considering the form of an input expression to determine the margin for a letter, you may want to enter the value in either inches or centimeters. You may also want to input the value explicitly (say 1.5 inches) or you may want to enter a calculation which produces the right input value (you know the width of the text is 6.5 inches and the width of the paper is 8.5 inches and you want the left margin to be twice as large as the right margin, so you enter 2/3 (8.5 − 6.5) inches). This input substitutivity contributes towards flexibility by allowing the user to choose whichever form best suits the needs of the moment. By avoiding unnecessary calculations in the user‘s head, substitutivity can minimize user errors and cognitive effort.

Robustness

A user is engaged with a computer in order to achieve some set of goals. The robustness of that interaction covers features that support the successful achievement and assessment of the goals.

Observability

Observability allows the user to evaluate the internal state of the system by means of its perceivable representation at the interface. Observability can be discussed through five other principles: browsability, defaults, reachability, persistence and operation visibility. Browsability allows the user to explore the current internal state of the system via the limited view provided at the interface. Usually the complexity of the domain does not allow the interface to show all of the relevant domain concepts at once. Indeed, this is one reason why the notion of task is used, in order to constrain the domain information needed at one time to a subset connected with the user‘s current activity. While you may not be able to view an entire document‘s contents, you may be able to see all of an outline view of the document, if you are only interested in its overall structure. Even with a restriction of concepts relevant to the current task, it is probable that all of the information a user needs to continue work on that task is not immediately perceivable. Or perhaps the user is engaged in a multi-threaded dialog covering several tasks. There needs to be a way for the user to investigate, or browse, the internal state. This browsing itself should not have any side-effects on that state; that is, the browsing commands should be passive with respect to the domain specific parts of the internal state.

The availability of defaults can assist the user by passive recall .It also reduces the number of physical actions necessary to input a value. Thus, providing default values is a kind of error

prevention mechanism. There are two kinds of default values: static and dynamic. Static defaults do not evolve with the session. They are either defined within the system or acquired at initialization. On the other hand, dynamic defaults evolve during the session. They are computed by the system from previous user inputs; the system is then adapting default values.

Reachability refers to the possibility of navigation through the observable system states. There are various levels of reachability that can be given precise mathematical definitions, but the main notion is whether the user can navigate from any given state to any other state. Reachability in an interactive system affects the recoverability of the system, as we will discuss later. In addition, different levels of reachability can reflect the amount of flexibility in the system as well, though we did not make that explicit in the discussion on flexibility. Persistence deals with the duration of the effect of a communication act and the ability of the user to make use of that effect. The effect of vocal communication does not persist except in the memory of the receiver. Visual communication, on the other hand, can remain as an object which the user can subsequently manipulate long after the act of presentation. If you are informed of a new email message by a beep at your terminal, you may know at that moment and for a short while later that you have received a new message. If you do not attend to that message immediately, you may forget about it. If, however, some persistent visual information informs you of the incoming message , then that will serve as a reminder that an unread message remains long after its initial receipt.

Recoverability

Recoverability is the ability to reach a desired goal after recognition of some error in a previous interaction. There are two directions in which recovery can occur, forward or backward. Forward error recovery involves the acceptance of the current state and negotiation from that state towards the desired state. Forward error recovery may be the only possibility for recovery if the effects of interaction are not revocable (for example, in building a house of cards, you might sneeze whilst placing a card on the seventh level, but you cannot undo the effect of your misfortune except by rebuilding). Backward error recovery is an attempt to undo the effects of previous interaction in order to return to a prior state before proceeding. In a text editor, a mistyped keystroke might wipe out a large section of text which you would want to retrieve by an equally simple undo button. Recovery can be initiated by the system or by the user. When performed by the system, recoverability is connected to the notions of fault tolerance, safety, reliability and dependability, all topics covered in software engineering. However, in software

engineering this recoverability is considered only with respect to system functionality; it is not tied to user intent. When recovery is initiated by the user, it is important that it determines the intent of the user‘s recovery actions; that is, whether he desires forward (negotiation) or backward (using undo/redo actions) corrective action.

Responsiveness

Responsiveness measures the rate of communication between the system and the user.Response time is generally defined as the duration of time needed by the system to express state changes to the user. In general, short durations and instantaneous response times are desirable. Instantaneous means that the user perceives system reactions as immediate. But even in situations in which an instantaneous response cannot be obtained, there must be some indication to the user that the system has received the request for action and is working on a response. As significant as absolute response time is response time stability. Response time stability covers the invariance of the duration for identical or similar computational resources. For example, pull-down menus are expected to pop up instantaneously as soon as a mouse button is pressed. Variations in response time will impede anticipation exploited by motor skill.

Task conformance

Since the purpose of an interactive system is to allow a user to perform various tasks in achieving certain goals within a specific application domain, we can ask whether the system supports all of the tasks of interest and whether it supports these as the user wants. Task completeness addresses the coverage issue and task adequacy addresses the user‘s understanding of the tasks. It is not sufficient that the computer system fully implements some set of computational services that were identified at early specification stages. It is essential that the system allows the user to achieve any of the desired tasks in a particular work domain as identified by a task analysis that precedes system specification Task completeness refers to the level to which the system services can be mapped onto all of the user tasks. However, it is quite possible that the provision of a new computer based tool will suggest to a user some tasks that were not even conceivable before the tool. Therefore, it is also desirable that the system services be suitably general so that the user can define new tasks.

STANDARDS

Standards for interactive system design are usually set by national or international bodies to

ensure compliance with a set of design rules by a large community. Standards can apply specifically to either the hardware or the software used to build the interactive system. Smith points out the differing characteristics between hardware and software, which affect the utility of design standards applied to them:

Underlying theory Standards for hardware are based on an understanding of physiology or ergonomics/human factors, the results of which are relatively well known, fixed and readily adaptable to design of the hardware. On the other hand, software standards are based on theories from psychology or cognitive science, which are less well formed, still evolving and not very easy to interpret in the language of software design. Consequently, standards for hardware can directly relate to a hardware specification and still reflect the underlying theory, whereas software standards would have to be more vaguely worded. Change Hardware is more difficult and expensive to change than software, which is usually designed to be very flexible. Consequently, requirements changes for hardware do not occur as frequently as for software. Since standards are also relatively stable, they are more suitable for hardware than software.

A given standards institution, such as the British Standards Institution (BSI) or the International Organization for Standardization (ISO) or a national military agency, has had standards for hardware in place before any for software. For example, the UK Ministry of Defence has published an Interim Defence Standard 00–25 on Human Factors for Designers of Equipment, produced in 12 parts:

- Part 1 Introduction
- Part 2 Body Size
- Part 3 Body Strength and Stamina
- Part 4 Workplace Design
- Part 5 Stresses and Hazards
- Part 6 Vision and Lighting
- Part 7 Visual Displays
- Part 8 Auditory Information
- Part 9 Voice Communication
- Part 10 Controls
- Part 11 Design for Maintainability
- Part 12 Systems

One component of the ISO standard 9241, pertaining to usability specification, applies equally to both hardware and software design. In the beginning of that document, the following definition of usability is given: Usability The effectiveness, efficiency and satisfaction with which specified users achieve specified goals in particular environments.

Effectiveness The accuracy and completeness with which specified users can achieve specified goals in particular environments.

Efficiency The resources expended in relation to the accuracy and completeness of goals achieved. Satisfaction The comfort and acceptability of the work system to its users and other people affected by its use.

GUIDELINES

A major concern for all of the general guidelines is the subject of dialog styles, which in the context of these guidelines pertains to the means by which the user communicates input to the system, including how the system presents the communication device. Smith and Mosier identify eight different dialog styles and Mayhew identifies seven . The only real difference is the absence of query languages in Mayhew‘s list, but we can consider a query language as a special case of a command language Most guidelines are applicable for the implementation of any one of these dialog styles in isolation. It is also important to consider the possibility of mixing dialog styles in one application. In contrasting the action and language paradigms , we concluded that it is not always the case that one paradigm wins over the other for all tasks in an application and, therefore, an application may want to mix the two paradigms. This equates to a mixing of dialog styles – a direct manipulation dialog being suitable for the action paradigm and a command language being suitable for the language paradigm. Mayhew provides guidelines and a technique for deciding how to mix dialog styles.

Smith and Mosier [325]	Mayhew [230]
Question and answer	Question and answer
Form filling	Fill-in forms
Menu selection	Menus
Function keys	Function keys
Command language	Command language
Query language	–
Natural language	Natural language
Graphic selection	Direct manipulation

Table: Comparison of dialog styles mentioned in guidelines

GOLDEN RULES AND HEURISTICS

Shneiderman's Eight Golden Rules of Interface Design They are intended to be used during design butcan also be applied, like Nielsen's heuristics, to the evaluation of systems.

1. Strive for consistency in action sequences, layout, terminology, command use and so on.

2. Enable frequent users to use shortcuts, such as abbreviations, special key sequences and macros, to perform regular, familiar actions more quickly.

3. Offer informative feedback for every user action, at a level appropriate to the magnitude of the action.

4. Design dialogs to yield closure so that the user knows when they have completed a task.

5. Offer error prevention and simple error handling so that, ideally, users are prevented from making mistakes and, if they do, they are offered clear and informative instructions to enable them to recover.

6. Permit easy reversal of actions in order to relieve anxiety and encourage exploration, since the user knows that he can always return to the previous state.

7. Support internal locus of control so that the user is in control of the system, which responds to his actions.

8. Reduce short-term memory load by keeping displays simple, consolidating multiple page displays and providing time for learning action sequences.

Norman's Seven Principles for Transforming Difficult Tasks into Simple Ones

1.Use both knowledge in the world and knowledge in the head. People work better when the knowledge they need to do a task is available externally – either explicitly or through the constraints imposed by the environment. But experts also need to be able to internalize regular tasks to increase their efficiency. So systems should provide the necessary knowledge within the environment and their operation should be transparent to support the user in building an appropriate mental model of what is going on.

2. Simplify the structure of tasks. Tasks need to be simple in order to avoid complex problem solving and excessive memory load. There are a number of ways to simplify the structure of tasks. One is to provide mental aids to help the user keep track of stages in a more complex task. Another is to use technology to provide the user with more information about the task and better feedback. A third approach is to automate the task or part of it, as long as this does not detract

from the user‘s experience. The final approach to simplification is to change the natureof the task so that it becomes something more simple. In all of this, it is important not to take control away from the user.

3. Make things visible: bridge the gulfs of execution and evaluation. The interface should make clear what the system can do and how this is achieved, and should enable the user to see clearly the effect of their actions on the system.

4. Get the mappings right. User intentions should map clearly onto system controls. User actions should map clearly onto system events. So it should be clear what does what and by how much. Controls, sliders and dials should reflect the task – so a small movement has a small effect and a large movement a large effect.

5. Exploit the power of constraints, both natural and artificial. Constraints are things in the world that make it impossible to do anything but the correct action in the correct way. A simple example is a jigsaw puzzle, where the pieces only fit together in one way. Here the physical constraints of the design guide the user to complete the task.

6. Design for error. To err is human, so anticipate the errors the user could make and design recovery into the system.

7. When all else fails, standardize. If there are no natural mappings then arbitrary mappings should be standardized so that users only have to learn them once. It is this standardization principle that enables drivers to get into a new car and drive it with very little difficulty – key controls are standardized. Occasionally one might switch on the indicator lights instead of the windscreen wipers, but the critical controls (accelerator, brake, clutch, steering) are always the same.