CHAPTER 3

Internal Properties: The Software Developer's Perspective

3.1 Introduction

Every engineering project is driven by the need to produce an acceptable product which matches the users' requirements and which will therefore be accepted in accordance with contractual obligations. Where a product is being produced speculatively in the hope of attracting users, there is just as strong a set of requirements (including costing and timing) as when a specific client has ordered something.

These general rules apply just as much to the developer of a software engineering product as to a civil or electrical engineer, the sole practical difference being that a larger proportion of software products is produced on a speculative basis. This makes (potential) customer involvement even more important; a feature of software engineering design which should be welcomed by a good professional engineering team.

There is, however, a very important technical difference between most software engineering products and most 'hardware' engineering products. Where hardware is concerned, the materials available for the product provide a limitation on what can be made. Unfortunately for the software engineer the investment of sufficient resources (including time) can nearly always achieve a product which is almost indistinguishable from the users' ideal. Most software engineering organizations, however, will wish to minimize the resources required to produce and support speculative projects to avoid exhausting resources, before the end product is completed and earning revenue.

As discussed in Chapter 2, customer satisfaction is provided when system behavior as perceived by the user is acceptable. The design of an interactive system, however, must also take into account other considerations which, in general, cannot be perceived or inferred by a user. For example, the user is not concerned with the designer's problems or the construction cost of a system (although perhaps with the price charged!) Also, while the user is likely to be directly concerned with the lifetime of the system when produced, difficulties of maintainability are only of indirect concern where, for example, a modification may turn out to be too late or too expensive – or both.

C. Gram et al. (eds.), *Design Principles for Interactive Software*

Designers' problems, which the user should not need to be aware of, include, for example, the difficulty of actually constructing the desired system and determining the actual effectiveness of the end result. Considerations of this kind necessarily affect the software and hardware architecture chosen, which, in turn, influences how the desired user-detectable properties are to be achieved.

This chapter therefore introduces and discusses those software engineering considerations which affect the construction and usability of an interactive system. The developer must look at several attributes which are neither observable, inferable nor measurable by users, but which influence the effectiveness of the development process and of the final result. These attributes, which are not visible to users, are given the collective term *internal properties.*

Internal properties are quality attributes of a system as seen from the developer's perspective, just as the external properties discussed in the previous chapter are system quality attributes as seen from the user's point of view. While several internal properties apply generally to all systems, the discussion in this chapter is confined to the user interface system architecture and those software engineering practices – software techniques – which relate to this.

The approach that will be followed, is to consider, from the designer's viewpoint, those software techniques which should be adopted to best satisfy the software quality goals throughout the entire life cycle of a system – from the first gleam in the designer's eye to the final system's demise. It will cover design and development methods for software creation, different approaches to the content of software, and will also discuss the application of software tools in order to produce the desired content. Together these three facets of a designer's work (methods, software content and tools) may be termed software techniques for the interactive system designer.

It should be noted that several different techniques may contribute to the achievement of any one internal property. The decision to make use of a particular technique to achieve one property may, however, have the side-effect of making it more difficult to achieve some other property or properties. The existence of such negative effects makes it essential to study the inter-relationships between properties and those software techniques which may be adopted to ensure that necessary quality goals are achieved. This is, of course, independent of whether the internal quality goal is set because of a user requirement or is an imposed development constraint. Imposed development constraints necessarily limit the design space available to the software engineer, giving rise to the need to consider additional design trade-offs.

There are many forms of interrelation between internal properties and software techniques. These are discussed in more detail in a subsequent section, after first covering the internal properties and then selected soft-

ware techniques. These software techniques also interact with the external properties from the previous chapter. This chapter closes with an analysis of these interactions.

3.2 Internal Properties

Internal properties require a complete life cycle view. It is important to recognize that these properties are relevant from the conception of a system, beyond construction to modification and maintenance until its final demise. Many properties are in a sense 'post manufacture' issues – such as modifiability and maintainability – and are sometimes neglected by developers. But the user interface is frequently the most highly modified portion of a system after its initial production, and therefore the consideration of *all* the issues is very important to the interface system software engineer. Normal operation must also be considered, even when no modifications or maintenance are required, since systems must not make excessive demands on processor power or storage.

We have selected eight internal properties that are particularly relevant to the development of interactive systems:

I1. System Modifiability – is the system easy to modify when it becomes desirable to extend its life or enhance its facilities?

I2. Portability – this must be viewed from three points of view: change of hardware environment, change of software environment and moving a user to a different environment using the 'same' system. How difficult/easy will these be?

I3. Evaluability – how easy is it to evaluate the system against quality goals (such as performance and suitability for new/different users)?

I4. Maintainability – once installed in a certain environment, will the system be easy to maintain (and manage)?

I5. Run time Efficiency – does the system use an acceptably low fraction of computer system resources in relation to the functionality it provides?

I6. User Interface Integratability – how easy is it (will it be) to integrate the interactive system with existing or new user software applications?

I7. Functional Completeness – does the system have sufficient functionality to support the users in solving their tasks – and to do so correctly?

I8. Development Efficiency – is the most effective use being made of resources during design and construction?

Two of these properties, Modifiability and Maintainability, may appear

to be very similar, but we still distinguish between them: a new task requirement or a change in the environment or the platform for the system is met by modifications in the product; maintenance is the work needed to keep a given system running in a given environment.

3.2.1 Modifiability

Once an interactive system has been released as a product, new or additional requirements may arise. This leads to necessary modifications (or new versions) of the product. The ease with which the system may be modified is a very important factor in improving life cycle effectiveness.

In practice the user interface is the most highly modified portion of an interactive system. This is one of the prime motivators for the development of the user interface software architectures discussed in the next chapter. Modifiability is influenced by several different factors:

- available development environment;
- target environment;
- re-use of existing specifications and code;
- separation of concerns – the ability to provide clean abstractions (and well defined interfaces) for system components;
- software architecture – the (re)composition of system components.

A typical development environment offers the designer both a set of tools to assist with specification and implementation and also a library of already designed and tested software abstractions (modules). The contents of the library may be at specification level and/or implemented code level.

The value of such library facilities strongly depends on how well the modules are parameterized. A well-parameterized module will offer the greatest flexibility; a poorly devised module will be relatively inflexible and may actually hinder modifications. Where a well-designed library of code exists, modification may be effected by amendment of the actual parameters used by a module or by changing one or more modules. In either case, existing code is re-used. If system modifications cannot be effected by re-using elements from a library, then new code will need to be constructed – with the additional development tasks of documenting and testing. Inevitably such new code will be more liable to failure than library facilities as it will have had less testing than code already used in other systems.

The target environment, in which a system will be used, offers both hardware and software facilities, together with inevitable constraints (e.g. good graphical user interface (GUI) support may be available, but little support offered for multi-media style interfaces). The ease or difficulty of modification is therefore strongly influenced one way or the other by the target environment.

In the design of any interactive system there will be sections or modules, which the designer's experience may indicate as likely candidates for future modification. Such anticipated modification will be more easily effected if the principle of separation of concerns is followed in the original design. Those sections for which modification is considered likely should (as far as practicable) be separated into pure abstractions. If the separation is well implemented, it results in the modification having no effect on other interacting components. Hence the development environment must contain tools that support good abstractions and provide facilities for generating re-usable code-modules.

Because it is highly likely that modification will be needed both within the user interface portion of the system and in the functional core, the ability to produce separately generated modules is an essential adjunct to ease of modification. But the full benefit can only be achieved if the overall software architecture supports (or even forces) such separation into modules. This separation and consequent encapsulation of functions not only assists with future modification, but also with the ease of eventual maintenance. This is further discussed in Section 3.2.4.

3.2.2 Portability

A system is said to be portable if it is easy (hence cheap!) to move it to a different environment. Three kinds of portability may be involved:

1. Change of target hardware – the hardware platform on which the system runs is changed but the system should still behave the same for the user. Such changes may have profound effects on the user interface and functionality of the system.
2. Change of target software – the software environment in which the system runs is changed (perhaps as an upgrade) and the system should still behave in the same way for the user. Such software changes occur frequently with little or no warning. Depending on the kind of upgrade, such changes may have profound effects on the interface.
3. Move of user – a user moves to another department while undertaking the same tasks. Differences in the platforms (for example due to small differences in versions of terminals,* file servers, etc.) must not show up as differences in the user interface. The system must supply the user with the same facilities and functions as before, even if the new platform is different from the one previously used.

The first two kinds of portability cover situations where the target platforms are changed, e.g. because new versions of hardware or software are

* We generalize the idea of a 'terminal' to any grouping of interactive input and output devices and the associated software (e.g. a complete workstation).

installed. The third kind covers a different, although superficially similar, situation, where a user moves from workplace to workplace expecting to be able to use the same system the same way with the same results.

Changing the hardware on which a system is to run is perhaps the most obvious form of portability, for example changing from a bit-mapped graphics display to a vector display or motion video device. It should not be forgotten, however, that 'merely' substituting a more modern version of some workstation may not be without hardware problems, where manufacturers have failed to adhere to previously adopted standards, causing unforeseen hardware/system incompatibilities.

The most difficult portability problem is posed by the need to maintain the same software interface following changes to the environment upon which it is built. Such changes occur frequently with little or no warning (such as installing an upgrade to an operating system) and can have profound effects on the running software product. The need to adhere to standards is even more important in this case. If such major portability problems are to be avoided, the system designer must be absolutely sure that his or her user interface does not rely upon non-standard features of some environment.

The third kind of portability, move of user, could be dismissed as an issue of administration only. It may also be considered a combination of the first and the second kind of portability. But it does cause problems sufficiently often in practice that the designer should prepare for it. The system must be designed for all the (slightly different) configurations and must be tested on all terminals linked to the system.

All three forms of portability, however, are special cases of modifiability. When discussing modifiability the focus point is the ease of changing the behavior of a system in response to a need for enhanced or changed user facilities. When considering portability, however, the items that have been changed are the user's place or the system's platform rather than functional requirements. Here it is a question of preventing changes in the environment from affecting the way the system behaves for the user. The modifications needed are to maintain previous behavior; not to permit changes from affecting the way the system behaves or performs for the user. In summary, modifiability addresses design changes, whereas portability occurs later in the life cycle during installation and operation. It has the aim of *preserving* the original design in the face of a new and potentially uncooperative target environment.

3.2.3 Evaluability

A system is said to be evaluable when it is easy to evaluate, whether or not it fulfills some specified quality goals. One method for enhancing evaluability is to build into the system facilities for obtaining metrics of various

kinds, related to the detailed behavior and performance evinced in use. In many standard systems (like C or Pascal) tools are available – directly or indirectly – to obtain measurements related to software properties like effectiveness, efficiency, error visibility or maintainability.

But from a usability point of view it should also be possible to measure the usability properties discussed in the previous chapter – predictability, migratability, etc. This requires careful consideration during design, as the evaluability is intimately connected with the facilities available in the development and run time environments. The run time facilities may include logging tools specific for each level of abstraction, producing reports such as:

- Physical-interaction-level logs capturing time-stamped patterns of users' keystrokes, mouse clicks, etc., used to assess low-level time-dependent external properties.
- Functional-level logs capturing each function invocation and completion, used to assess, e.g., pace tolerance and run time efficiency. The logs also reveal functions seldomly used (candidates for elimination) and functions that often fail (candidates to be rewritten with improved error checking).
- Dialog-level logs capturing patterns of user–system interaction, used to assess external properties, such as non-preemptiveness, insistence and deviation tolerance.

Irrespective of the actual development and run time facilities, the development approach used (prototyping, incremental development) should provide for taking such measurements and incorporating the results of the evaluation into the system as it is being developed.

3.2.4 Maintainability

Maintenance is that effort which is necessary to keep a given system running in a given environment (in contrast to modifiability which measures the work to include *new* functions in a given system).

Maintenance includes system administration; installation of new printers, displays, etc.; tuning of the system and error correction. Tuning and error correction together make up typically only 20% of the work while administration and hardware adaptation swallow 80%. The administration work comprises things like version control, library updating and re-installation (with altered set-up, or with a new window manager, etc.).

A system is said to be maintainable if:

- a system administrator has an easy job keeping the system running;
- the existence of errors which could cause failures is easily detected and, when failures do occur, those errors are easily corrected.

System administration is helped if the system is clearly structured, and it is systematically and accurately documented. This is most easily encompassed as part of an overall quality assurance plan. Maintenance should also be supported by good tools (software packages) for version and library control, etc. Providing facilites for monitoring system behavior in response to user interactions could also offer help in determining load patterns, bottlenecks, most frequent kinds of user mistakes, etc.*

A software system differs from a hardware product in that it is not subject to wear. However, if over-stressed (not used in accordance with its specification) it can suddenly break down just like a piece of hardware. Just like hardware, too, errors which cause failure are present from the day of manufacture, but do not reveal their presence until a user performs a seldom exercised function or misinterprets some system response.

The difference between user interface software and most other engineering products, therefore, is that maintainability measures need to include user mistakes (errors of misuse) as well as system errors. A user mistake may arise from two separate actions.

1. Because a task execution does not solve the user's task in the expected way. What the user believes to be a correct task step in the actual situation is invalid or leads to an unexpected result.
2. Because a system response is misinterpreted by the user, such that the user continues task execution 'in a wrong direction'.

Errors may be caused by the underlying operating system and hardware, by the user application or by the interface system itself. The first two of these error sources behave unpredictably from the point of view of the user interface system. As such it is extremely difficult to design for maintainability in respect of them.

For the user interface system, however, reduction of number of errors in the first place and ease of error correction improves maintainability. High error rates are likely to result in requests for change. Developers must therefore strive to prevent errors and to make it easy to correct errors. A principal means of doing this is by re-using code as much as possible and by making use of standards and standard development toolkits. The resulting consistency aids users in learning how to correct basic input errors, as well as reducing the need to learn new interaction techniques, which would increase errors during the learning period. However, this tactic only addresses the logical level of interaction, with occasional standardization of dialog fragments (e.g. select command, fill in dialog box, accept dialog box). The provision of tools and materials for supporting deviation tolerance at all levels of interaction is discussed further in Chapter 5.

* It must be mentioned in this connection, however, that any such monitoring may be in conflict with local privacy legislation.

3.2.5 Run time Efficiency

While other components of performance are relevant in an overall sense, the most important measure of run time efficiency for an interactive user interface is the response time of the system to user input. This is influenced by a variety of factors, including:

- the software architecture adopted;
- the algorithms and heuristics which have been incorporated;
- the underlying software and hardware.

It is unfortunate that, in general, run time efficiency is reduced by the adoption of mechanisms to alleviate some of the other problems of the interactive system developer. For example, improving deviation tolerance by the provision of undoing can make extensive demands on storage space, especially when unlimited backtracking is supported. Thus, the process of system design necessarily includes making trade-offs between run time efficiency and some of the other problems discussed.

3.2.6 User Interface Integratability.

The typical user has several activities to perform and uses a number of different interactive systems. This means that a new interactive system must ideally integrate transparently with the existing user facilities in the following ways.

- The interface of the new system must not be significantly different in apparent behavior from existing systems. The reason for this is that users may well continue to use existing systems at the same time. The new system must therefore work in a manner which is intuitively the same as existing software in the workplace, so that users can move seamlessly without difficulty between the old and the new. It must at least provide the required functions in a way which is not counter-intuitive to the user of other software. For example, the functionality assigned to picking devices (e.g. mouse button mappings) must not be different, and in the 'File' and 'Edit' menus, options such as those for saving files and copying to the clipboard should use the same names, short cuts, and other menu features.

  These requirements are closely associated with some of the criteria for portability (Section 3.2.2) and predictability (Section 2.4.4).

- One of the crucial aspects about the introduction of a new interactive system is its ability to work correctly with existing software. The new system must, therefore, interface to existing software applications so that they – at the functional and the dialog levels – behave identically to the way they have always behaved in the past in spite of the new interface.

- The new system must not disrupt the target software or hardware environment in such a way that the behavior of other existing software is affected perceptibly. That is, the new system must not use the resources in such a way, that any of the other (independent) programs are impeded perceptibly.

Interface integratability in these ways is more easily obtained the more the developers are able to adopt relevant standards both for the interface software being built and for communications between different application systems. These forms of interface integratability become progressively easier to achieve when developers make increasing use of application design standards (e.g. Windows Application Design Guide, Microsoft (1992)) and inter-application communication standards – two software techniques that influence the satisfaction of internal properties, as discussed in Section 3.4.

3.2.7 Functional Completeness

The reason for constructing a particular system in the first place is to satisfy a set of task requirements. The external property of task completeness requires that designers describe the necessary interactions for all identified tasks. At the functional level, task executions involve the application of abstract commands to functional state elements. During construction, developers must find ways to implement these abstract commands and functional state elements.

A system is *functionally complete* if developers can faithfully implement all the abstract commands and functional state elements required to support all identified tasks. Functional completeness is thus conformance to the specifications that result from earlier task analyses.

The ease (or difficulty) with which this completeness may be achieved is therefore a major concern for the development team. The proper choice of design, refinement and testing methods is consequently of great importance. Several software techniques have an impact on the achievement of functional completeness. The required functionality at some level of system abstraction may be given extensive support by the target environment, reusable code, or the I/O resource manager. But the capabilities of these fixed components may also prevent implementation of a required feature (e.g. early implementations of the X Window system could not support double-clicking at the logical interaction level). Similarly, user interface and inter-application communication standards may aid, impede or block the efficient and/or effective implementation of required functionality.

3.2.8 Development Efficiency

The efficiency with which it is possible to develop a system, must not be confused with the effectiveness of the design process or of the design itself.

Efficiency as used in this chapter is defined as making the best possible use of the resources available to the designer during development. To a large extent this is a concern for the project manager, but the developer must also be aware of the fact that methods and techniques selected for the design may influence the overall efficiency.

The entire development process includes construction and testing which, in common with most other branches of engineering, are often more labour intensive than the design process. It is principally, therefore, these phases which must be considered in attempting to improve efficiency. Development efficiency is thus related to the following principal factors.

- The complexity of the development methods used (e.g. iterative, predictive and experimental methods, see below).
- The development environment and tools available to the engineers.
- The software architecture being developed. If the architectural model does not easily fit with the systems requirements, compromises have to be made in the software architecture, and this may impede the development efficiency.
- The target platform for the product which may place more or less severe restrictions on the available options for implementing desired facilities.
- The need to adhere to published standards or local software engineering practices.
- The size and composition of the development team.

In order to permit the rapid development of sophisticated, highly dynamic user interfaces, there is a need for tools and techniques to assist the designer. At present such user interfaces are more difficult to design and implement than were command line interfaces. Certain *de facto* user interface standards can be used for more traditional interfaces, but for multi-media, time-dependent interfaces with audio input no standard is available. Those tools which are being developed to assist the software engineer (e.g. Visual Basic) are still experimental, and they only cover some of the possibilities (i.e. GUIs).

There is limited experience of developing such highly dynamic interfaces. There are, for these reasons alone, very few standards of any kind to which the engineer can adhere or even use as guidelines when designing. In the short term the developer must therefore develop highly dynamic interfaces in the absence of significant tool support for formal design, and in the absence of comprehensive standards (whether implemented in available standard software components or not).

To some degree, the absence of standards may hinder efficient development, because the development engineer must invent his or her own methods and rules. This may also hinder efficient re-use of already developed software. While adherence to standards (whether formal or local) will help

to reduce development design effort, as will the adoption of accepted engineering practice, this will only be of assistance as long as the standards and practices concerned suit development of the kind of system being produced. The problems of being constrained to adopt inappropriate standards will inevitably hinder, if not inhibit, satisfactory product development, resulting in functional incompleteness, inefficient development, or both.

Several different software architectures have been developed to support user interface development. The in-depth study of a representative selection of these is deferred to the next chapter. However, some initial observations can be made. When a system developer uses an existing software architecture as the basis for a new design, there are several advantages.

- The architecture has been analysed and its advantages and disadvantages are known and are documented.
- The architecture will usually have been tested in practice, and such tests will presumably also have been documented.
- The architecture may have been embodied in a development environment specialized for it and, consequently, sophisticated tools could be available to support the development of systems based upon it.

However there are also potential disadvantages which arise from using any pre-packaged architecture.

- The architecture was optimized to support features that may not be important in the current development.
- The architecture was similarly not optimized to support features that are important to the current design.
- The compromises inevitable in a general design could have severe implications for the simplicity with which complex requirements can be met in an effective and efficient manner.

The nature and facilities provided by the target environment platform will also affect development effort. The target platform will in almost all cases provide undesired constraints on such things as memory, processor power and peripheral channel performance. It may, probably far more importantly, not use the same software environment as that being used by the development platform.

Lastly, the size and composition of the development team is a primary determinant of development efficiency. The more experienced are the members of the team in developing systems similar to the current design, the more efficient will the development process be. On the other hand, development efficiency is inversely dependent on the size of the development team because of the communication requirements engendered by multiple developers. The more developers, the more difficult it is to maintain useful communication.

3.3 Software Techniques

When discussing external properties in the previous chapter, frequent reference was made to those design features that aided or impeded the satisfaction of an external property. In the above discussion of internal properties, software phenomena which aid or impede the satisfaction of an internal property have also been mentioned. These phenomena may be collectively grouped together under the heading of *software techniques.*

Each internal property may be achieved – at least to some degree – by the judicious application of one or more software techniques. This section discusses those software techniques which are seen as particularly appropriate to interface system design and building. The following section discusses their applicability under particular circumstances.

Software techniques take many forms (this is why the term is used in a loose sense). The main forms are: methodologies, tools and standards. Methodologies provide guidelines for the development of software systems. Tools generate or analyse (components of) software systems. Standards provide guidelines for the behavior and other features of (components of) software systems. The techniques that are considered most relevant to the design of interactive systems fall into these three groups as follows.

Methodologies used as guidelines during the development of the interface system:

1. **User interface design methods**
2. **Architectural modeling**
3. **Global software re-use**
4. **Quality assurance planning.**

Design and implementation tools to generate or analyse parts of the system:

5. **Specification languages and tools**
6. **Input/output resource management tools**
7. **Target environment facilities.**

Standards that provide definitions and guidelines for behavioral and other features of the system:

8. **User interface standards**
9. **Inter-application communication standards.**

All the above techniques have been referred to during the preceding discussion of internal properties. The list given is not intended to be exhaustive, but rather to support the general approach adopted for this book – to highlight, exemplify and analyse relationships between diverse aspects of software quality and elements of the design, such as separation of concerns,

composition principles, and encapsulation. In Chapter 4, the software techniques are considered in the light of software architecture models, and in Chapter 5 the techniques are related to the development process.

As noted at the start of this chapter, the employment of software techniques applies to entire software systems, but the discussion here is restricted to consideration of user interface aspects.

3.3.1 User Interface Design Methods

The intuitiveness of a particular user interface for users, and the effectiveness with which it can be used are very difficult to predict. First of all, the design process should be helped by clearly distinguishing the four levels of abstraction introduced in Chapter 1: the functional, the dialog, the logical interaction and the physical interaction levels.

In response to the difficulty of prediction, most user interface design methods include the use of evaluation techniques to gather early feedback from other specialists or from representatives of the user community. Combining this with the fundamental principles of iteration referred to in Chapter 1, this may be restated as:

- *Iterative design*, where each development version of the system is evaluated (by users and others), and evaluation results are used to design the next version. A special version of this is *prototyping*, i.e. the rapid construction of a portion of the user interface with limited, simulated or non-existent functionality. A prototype may be constructed by hand or by a tool able to translate a specification into executable code.

A number of evaluation techniques are widely used. It is necessary to distinguish between (i) predictive methods that can be used very early in the design phase of a project (i.e. during the specification phases, as soon as a specification or even a low-tech prototype is available), and (ii) experimental methods where some version (prototype) of the system is used. Some widely used evaluation techniques may be classified as follows.

- *Predictive methods* applicable early in the development process:
  - HCI-based design heuristics, such as:
    Principle-based Inspection – inspection by specialists for certain technology aspects, such as non-preemptiveness, observability, etc.;
    Style conformance inspection – inspection by specialists for conformance with published style guides such as the Windows Application Design Guide (Microsoft, 1992).
  - Cognitive-theory-based methods, such as:
    Cognitive Walkthrough – inspection by specialists for learning problems, such as operation visibility, honesty, etc., discussed by Polson *et al.* (1992);

GOMS method – use of a cognitive model using Goals, Operators, Methods and Selection rules, for a system to evaluate the efficiency and/or learnability of the dialog.

- Formal methods for assessing properties, such as using a *formal specification* of a dialog to prove that it has some specified properties (e.g. reachability).

- *Experimental methods* that require a running prototype or some mock-up of the system under development:
  - Participative design – presentation of the user interface and the functionality of the developing system to user representatives.
  - Summative evaluation – structured and planned evaluation of the finished product by usability specialists, with measurement against required targets.
  - Heuristic evaluation – informal but planned examination of whether the system fulfills a pre-identified set of heuristic usability criteria (Nielsen, 1992, 1993).
  - Usage observation – semi-structured monitoring and observation of real users' interaction with the system.

Once a prototype user interface has been built, it is imperative for the designer to obtain user opinion, even if predictive evaluation has been applied. The system must be tested by potential users. The purpose of such tests is to obtain both objective measures of user difficulties and subjective impressions from the user of ease of use, good and bad features, ease or difficulty of learning, etc. Such user tests need very careful design and preparation. The inevitable weaknesses of a prototype (with slow or missing facilities) may lead to user frustration if the test users are not suitably instructed.

In subsequent design work it must not be forgotten that the subjective impressions gained in such tests are potentially more important than actual measurements, since a prototype can rarely offer the same performance because of the general purpose nature of the tools used in its construction.

In contrast to a prototype system, a system functional walkthrough need not be conducted with a computer-based system. It could just as easily be based on low-technology prototyping such as flip charts, recorders or other presentation mechanisms. Recent developments in participative design have greatly extended approaches to low-tech prototyping (Muller *et al.*, 1993).

Whichever mechanism is chosen to derive user impressions and study the usability of the design, it is important that the process is not merely a single linear step in design. It may be necessary to iterate walkthroughs and prototype experiments, until both software engineer and users are content with the proposed design.

3.3.2 Architectural Modeling

Most systems used in the real world are so large and complex that it is impossible to grasp all details and to have a total understanding of the system and its functionality. The way to better understand complex systems is to use abstraction and to analyse simplified formal models of the systems.

The adoption of a high-level abstract architectural model for the design of an interactive interface cannot, therefore, be too highly commended. The model must provide a formal definition of an abstract solution to some set of design requirements that is sufficiently general to incorporate the principal requirements of the system being designed. Using such a model as a basis for detailed design has the immense advantage that the model's designers have carried out a full analysis to ensure its suitability for the specified range of system types. The developer using it therefore only has to carry out design analyses at the detailed level, if no major architectural modifications are found necessary.

The formal model of an interactive system must have the ability to capture not only functionality in the classical sense, but also the essential interactive nature of a dialog. A potential advantage of using such a formal model as the basis for an interface system design is that the formal tools which are becoming available could make it possible to not only specify the detail design formally, but also to provide a large measure of design verification automatically. This obviously is of great significance for the design aspects of quality assurance protocols as discussed below.

The following chapter examines a number of abstract software architectures suitable for the design of an interactive interface, elaborating on these principles.

3.3.3 Global Software Re-use

The re-use of functional components, originally written for use as part of other systems, is attractive from several points of view.

- The software exists and has (presumably) been tested in that earlier system. This reduces the errors inevitably inserted when building a new system.
- The cost of producing software for the new system is reduced by the effort that would otherwise be expended on design, building and testing of the component.

Along with the advantages there come responsibilities and, if these are ignored, some possible disadvantages.

- The design of the original component should have been set out as an abstraction (an abstract data type) so that its use depends upon nothing except itself and those items which were used in its original construction. This is a responsibility of the original designer/implementer.

- The documentation of the re-usable component must be complete and (preferably) formal so that there can be absolutely *no* misunderstanding about how it should be used in another environment.
- The implementation must have been designed and tested against the formal specification, otherwise it is almost worse than useless as it would have to be considered unsafe.

It is worth noting that several existing user interface toolkits have been successfully used (principally commercial or public domain windowing systems for bit-mapped displays). In fairness, however, it must also be pointed out that the majority of these have a limited software interface choice and have not yet been ported to a sufficiently wide range of programming languages to be of completely general applicability. Nonetheless the designer should strive to re-use existing software, because besides reducing development costs it may contribute significantly to the maintainability and modifiability of the system.

3.3.4 Quality Assurance Planning

The popular saying that 'a reputable manufacturer produces reliable products' is a tautology, because the adjective 'reputable' hides a great deal of conscious work and effort by the manufacturer to retain the (well-earned) reputation. Most of this effort is done in the names of quality control and quality assurance. These two complementary aspects of quality are both in their own way important to the ability to earn that good reputation.

Quality assurance (QA), the preventive medicine of quality, is the work done to ensure that the production tools and the production methods employed are all conducive to minimize errors/failings in the resulting product. Quality assurance is thus not related to any specific product or type of product, rather to the methods and techniques which are necessary to produce it. Careful QA planning is required to achieve reliable products.

Quality control, on the other hand, is the curative medicine, which has to be applied to test completed products in an attempt to 'prove' that the quality assurance procedures have succeeded, providing feedback for further improvement in them as and when needed.

Quality assurance comprises those procedures, protocols and records maintained in relation to the entire design and production work, which will provide early warning if something is not working correctly. Such a simple matter as recording not only every design change made, but also the reason for the change, will prevent extra work when some later decision would tend to reverse this decision without knowledge of the very likely unrelated reason for the earlier choice.

For the designer of interface software, the quality assurance protocols used vary little from those needed for other software, except that they

must cover the user tests, which constitute a very important part of the interface development. These tests will involve performance and subjective satisfaction targets that must be discussed, agreed and revised with user involvement. Users must thus form part of the QA mechanism when developing quality procedures.

The kinds of records and procedures needed for QA are carefully laid down now in both national and international standards (such as ISO, 1987). It is worth pointing out that the practice of quality assurance is frequently annoying in its inception due to the extra tasks and more rigid procedures which have to be adopted in working in the required way. But practical experience has shown that engineering firms which have adopted the formal mechanisms have reduced their costs in the long term and quality is indeed improved, sometimes to an astonishing extent.

It is important, however, to reiterate that preventive medicine is not foolproof and that quality control testing (and possible repair) of the end products themselves cannot be omitted. Doing so in a production environment which operates a quality assurance system offers much less costly 'restorative medicine' than would be required were final product testing the only means adopted for controling product quality.

3.3.5 Specification Languages and Tools

Those specification languages and tools chosen for use in any particular project are intended to simplify the eventual construction of the actual system by providing a formalism for specifying of the interactive system. In contrast to the results of using prototyping tools, which are intended to construct throw-away prototypes, the specification languages and tools are used to define and document the final executing system (or at least a part of it). The tools may help to check completeness and consistency of a specification, thereby supporting development efficiency. Without such tools, incompleteness and inconsistency may not be apparent until user testing. Making corrections at this late stage is bound to be more expensive: inappropriate features will have been implemented, and appropriate ones may have been omitted. Addressing the former involves throwing work away. Addressing the latter may involve expensive changes to the software architecture in order to accommodate the missing features.

In general, specification tools can improve general quality merely by letting problems be detected and addressed before the expense of construction and testing, by which time the resources needed for remediation may be unavailable.

A variety of such specification languages exists, together with a few tools for analysing specifications and for transforming specifications to generate final systems. Some of these will be discussed in Chapter 5.

3.3.6 Input/Output Resource Management

As will be described in discussing the use of inter-application communications, the problem of resource sharing of any kind brings with it – besides the request for data transparency – the need to ensure fairness and possibly mutual exclusion in the communication protocols.

Therefore, whenever multiple applications wish to share a resource (such as, for example, a bit-mapped graphic display) a resource manager should be used to coordinate and arbitrate between requests for access to that resource.

A principal feature of any such resource manager is that it needs to provide for an arbitrary number of applications and their interfaces requiring access to a single resource. It is important, therefore, that the manager provides timely response both to program requests and to external requests – it may usefully be thought of as a real time component in almost any workstation environment.

Another important feature is that a resource manager be entirely transparent to its clients. No one client should need to be aware of the existence of other clients unless, of course, there is a functional need for such awareness. Even then, the resource manager must not be overtly visible when inter-client activity is taking place through its mediation.

A third major requirement of a good resource manager is the separability of the management of the communications resource from the actual transfer of data via that resource. Such things as opening/closing connections and setting or obtaining connection status should be completely divorced from the transfer of data using that connection. This is best obtained by using sound abstraction principles in the design – separating resource management from data transfer.

To illustrate the complexity of I/O resource management, consider that part of an interface which is controling the current standard output channel from some application. This part of the interface may need to arrange for the channel to be connected to one of a number of devices (e.g. a display window, a loudspeaker, and a remote communications line) all at different times during one invocation of an application. This function can be abstracted as 'standard-output-channel' – only provided that the transfer of data through that channel can be done in a device-independent manner. If this cannot be done, the designer may have to use a completely different architecture – forced by the limitations imposed by the resource manager failings.

3.3.7 Target Environment

The choice of the target environment (in so far as it is under the control of the system developer) almost invariably affects the difficulty of both the design and the construction of an interactive system.

When the target environment is distinct from the development environment, some additional effort must be spent in order to ensure that the developed system operates correctly in the target environment. Such simple matters as the availability (or not!) of a particular keyboard key, or a monochrome target whereas the development environment had 24-bit color, may seem trivial, but can completely frustrate the user of what in prototype on the development hardware looked very good. The user tests performed during the development should always be carried out in the target environment.

The use of standard toolkits, window systems and resource (window) managers provides a choice of techniques to achieve this. Error detection and correction is also of much greater concern when the target environment is distinct from the development one.

In any case, careful selection of the components in the target environment is the preferred technique to help satisfy some of the software engineering problems discussed above, such as portability and functional correctness. For example, the mouse must work the same way (both speed and button use) in the development and the target environment; data buffers and swap areas must be large enough in the target environment to allow the same size and speed of data transfer as in the development environment.

3.3.8 User Interface Standards

The interface developer – like any software engineer – must take into consideration not just official standards, but also guidelines and common conventions. All standards develop from guidelines derived from conventions which in turn have been adopted as encapsulations of good engineering practice. It is important, therefore, to realize that the use of three terms really refers to the same concept at various stages of its life. That which is today's standard is yesterday's guideline and the previous day's convention!

The existence and content of a standard is, for similar reasons, changeable as further technical knowledge or insight is gained over time. It is most important, therefore, that a standard is not treated as a constraint by the software engineer. Its existence merely confirms that a large number of experts have come to an agreement over the standard after several years of discussion, but like almost everything else in the computing world the advance of technology encourages the amendment and improvement of standards as understanding improves and ideas develop.

The interface developer must consider standards at two levels:

1. Internally within the system.
2. Externally in interaction with the end user.

At the system level, the adoption of standards offers the advantage that the designer is given some interface specifications rather than having to develop them from scratch. Thus, various portions of the system can be integrated more easily than would otherwise have been the case.

At the external interface, end users can be given a (standard) interface style with which they may be expected to be familiar. This decreases training time for a particular system where the same style has been used previously. A further advantage of using standard interaction styles is that the standard interfaces can be tested for usability in a general setting rather than replicating some of the testing for each system.

In both cases, the adoption of standards potentially leads to the development of re-usable software components which implement a specified functionality. The topic of developing and using such re-usable software is discussed further below.

Where the adoption of one or more standards has been specified as a requirement, it must be realized that they *may* act as a constraint dependent upon the appropriateness of the choice made in relation to the interface system being designed. This is not, of course, certain. The potential for constraint which may be engendered arises from the necessary nature of a standard – it offers the solution of some less specific problem.

The underlying assumption in the considerable effort expended in developing all forms of standards is that such general solutions provide an organization with wide advantages, even though they may not be optimal for every (or indeed any) system. A wise selection of the appropriate standards to be adopted in helping to solve a particular problem will minimize the disadvantages while maximizing the advantages to be gained from their adoption.

3.3.9 Inter-application Communication Standards

An important feature of the software running in any modern computer system is the ability for applications to share both control and data. A typical example of such sharing is the ability to pass data from one application to another under control of the user. This kind of *inter-operability* requires that the inter-application communication follows some standard rules.

- The two applications concerned share a common form of data representation.
- The facility offered for a user to move presented data (sharing a device between two applications) requires that applications share the services of a device to 'move' the data (for example, between one display window and another).

- The protocols that are used to effect the data movement must be defined in common by the two applications involved. As an example, the editor functions cut, copy, paste, send, receive must operate in a common context for several applications. As another example, the co-existence of multiple window managers requires that they co-operate to share the display resource.

Even if the same data representation may not be suitable for all the applications involved in this kind of interaction, all those involved must agree on a representation for data exchange. Similarly, the protocol interaction must be dealt with by the underlying software in a uniform way such that different applications can react coherently on receipt of a message. The use of inter-application communication standards is a mechanism that allows this kind of data and protocol interchange.

3.4 Internal Properties and Software Techniques

The preceding sections have listed a number of the principal software engineering problems – related to internal properties – faced by the developer of an interactive system, together with some techniques which could be used in attempting to solve them.

Achievement of the internal properties mentioned can be influenced by several (or all) of the techniques in some way or another. This section outlines the most important relationships between internal properties and techniques. Table 3.1 gives a summary of the relations. The following discussions for each internal property describe how each problem may be alleviated by a proper combination of development techniques and amplify the table in respect of the matters discussed earlier in this chapter, pointing out the specific interactions which must be considered by the design engineer.

I1. Modifiability. User interface software architectures are designed to support the modifiability of the user interface. Thus, the adoption of one of those architectures already developed and formally analysed will improve the modifiability of the total system. The use of specialized specification languages and tools will reduce the effort needed to modify a system, because a formal description is easier to manipulate than an informal one. To some degree the systematic re-use of code and the proper choice of target environment may promote modifiability.

Table 3.1 *Relationships between software internal properties and software techniques in the design of interactive systems. Entries indicate the relations:*
++ : is a primary mechanism to meet the concern
+ : has a secondary effect on solving the problem
- : may have a negative effect on solving the problem
empty: the technique has no significant effect with regard to the property

	The Use of Software Techniques								
Internal Property	Design methods	Architect. models	SW Re-use	QA Plann.	Specif. Lang.	I/O Res. Manag.	Target Envir.	UI Stand.	Comm. Stand.
I1 Modifiability		++	+		+		+		
I2 Portability					+		+	++	
I3 Evaluability	+			++	+				
I4 Maintainability		+	++	+			+	+	
I5 Run time Efficiency		+				+	++		-
I6 UI Integratability	+		+				+	++	++
I7 Fct. Completeness		+/-	+			+	+	+	+
I8 Dev. Efficiency		+	+	-	++				

I2. Portability. Provided that the target hardware can offer the required functionality in some form it should always be possible to port a system to that hardware from some other hardware. This does not, of course, imply that the functionality will be identical or even acceptable – nor does it imply that porting is easy. Problems in this area relate to such things as requirements for a rich display color spectrum for correct functioning, speed penalties with an inadequate central processing unit (CPU), or unacceptable performance of some disc storage.

The adoption of software standards and guidelines for a target environment will have a major impact on the possibility of moving to a different platform with minimal effort. The availability of standard software protocols, interfaces and facilities ensures that the required functionality is standard across platforms to a greater extent than would have been the case, had such standards not been available and adopted.

The use of specialized formal specification languages and tools helps to make the interface system independent of the target platform. The platform dependency then becomes a function of the conformance to standards of the compilers, tools and resource managers – not of the system being built. Adoption of such formal techniques should also, therefore, enhance the portability of the interface system.

I3. Evaluability. This is enhanced by adopting and planning for the use of user interface design methods such as walkthroughs, since evaluation of behavior is the prime purpose of employing such methods. A steadfast intention to perform predictive testing will ensure that the system is evaluated in early phases of development. The presence of assertions and conditions in formal specifications allows these to be converted into run time instrumentation of the interface, which can again assist in evaluation. The use of such techniques, therefore, also supports the satisfaction of this property.

Most important of all, however, is the adoption of quality assurance protocols and methods. Recording all measurements, decisions and reasons – and having formal change mechanisms and review processes during system development – prepares the ground for evaluation and for exploitation of the evaluation results.

I4. Maintainability. Several software techniques may improve the maintainability of a system. If, in the design, a well-structured architecture is chosen, with loose couplings between components and strong cohesion in each component, it will be rather easy to implement the necessary maintenance. Systematic quality assurance planning ensures that full regression testing is carried out when updating and changing the system.

The systematic re-use of code in an organization can be a major contribution towards improving the maintainability of the systems developed.

Maintainability is also enhanced through the use of standards embedded in software toolkit components, since they are used in a wide variety of environments and consequently may be expected to have been thoroughly tested. It may be further enhanced through having a separate target environment from the development environment, since different platforms tend to exercise software behavior in different ways, exposing different errors.

I5. Run time Efficiency. The key factor in providing an efficient run time performance is the choice of target platform. Usually, the more resources available in the target environment, the *faster* the interactive software and the application programs will execute. But the development engineer must be concerned with efficient use of the available resources whatever they are. Here the choice of an appropriate architectural model may enhance efficiency (although few user interface architectures focus on efficiency, most introduce additional overhead).

The use of a resource manager should increase efficiency, since such managers have been carefully optimized for efficiency, and therefore tend to be more efficient than any individual system design will be. The adoption of standard inter-application communication protocols in many cases introduces overheads because they involve additional data transformations, lowering run time efficiency.

I6. User Interface Integratability. The developer often works under the constraints of what target environment is available to the users. But given this constraint, the adoption of appropriate standards, whether for user interfaces or inter-application communications, will ensure as far as possible that a new interface integrates smoothly into the user's workplace. It is most important that any new interface is not incompatible operationally with the other systems with which a user is familiar.

The use of prototyping/walkthrough techniques and their embodiment in the quality assurance protocols also improves integratability of a new system into the user environment. By letting users examine prototypes, designers can discover features of a new system that are incompatible with existing interactive software at the user site. It must be ensured that the users' feedback is taken formally into account in the system design process. This may do more than anything else to produce the desired end result.

Finally, the re-use of standard software components is also likely to give the end user a look and feel with which they may be expected to be familiar, provided that their existing environment supports these components.

I7. Functional Completeness. The ease with which the functional requirements can be met depends on the target platform, that is, the

target environment components, and additional materials such as I/O resource managers and libraries that implement standards. These, therefore, must be selected appropriate to the functional needs. There is no primary mechanism for achieving the property, and several of the software techniques considered here may further the achievement of functional completeness. But architectural models can aid or impede the provision of advanced user interface support facilities as discussed in Cockton (1991).

I8. Development Efficiency. The use of languages and tools specifically designed to specify user interfaces has a significant influence on the development efficiency. It may significantly reduce the time required to produce and prove a specification. The use of an already analysed software architecture will also help, since a portion of the high-level design has been completed. Similarly, the re-use of existing code can mean that the functionality embodied in that code does not have to be re-designed, re-built and re-tested. Lastly, the current use of predictive user interface testing can require extensive developer effort with uncertain gains in terms of design improvements. This can have a negative impact on development efficiency.

3.5 External Properties and Software Techniques

Previous sections have discussed a number of techniques available to the developer of an interactive system and stressed important relations between the techniques and some of the major concerns for the software engineer. The previous chapter introduced a number of important 'external' properties of user interfaces. Interactions between these external properties and software techniques can now be explored. In no case does the application of a technique ever guarantee that the constructed system has a specific property, but certain forms of some techniques can aid or impede the satisfaction of an external property.

Each user interface property is discussed in this section from this point of view in relation to the software engineering techniques outlined in this chapter. The discussion is summarized for flexibility properties in Table 3.2 and for robustness properties in Table 3.3.

3.5.1 Flexibility Properties

The use of good user interface design methods and the use of a well-structured architectural model are of great importance for almost all the flexibility properties, as indicated in the first column of Table 3.2. Each of the other software techniques influences some of the properties as discussed below.

Table 3.2 *Relationships between flexibility properties and software techniques. Entries indicate the relations:*
++ : is a primary mechanism to help achieve the property
+ : has a secondary effect in achieving the property
- : may have a negative effect on achieving the property
empty: the technique has no significant effect with regard to the property
'Customizability' covers here both Reconfigurability and Adaptivity

	The Use of Software Techniques								
Flexibility Property	Design methods	Architect. models	SW Re-use	QA Plann.	Specif. Lang.	I/O Res. Manag.	Target Envir.	UI Stand.	Comm. Stand.
Device Multiplicity	+	+				++	+		
Representation Multiplicity	++	++							
I/O Re-use	+	++						+	++
Role Multiplicity	+	+				+			++
Multithreading	+	+			++	+	-		
Non-preemptiveness	+			+	++	++	-		
Reachability	+				++				
Customizability	+	++			+/-	+		-	
Migratability	++	++			-			+	+

Device Multiplicity

The ability of an interface system to provide flexible use of multiple input/output devices is related to the use of a resource manager and the facilities which it provides. Devices and channels are controlled and provided by the resource manager, as a facility for the user interface system. The importance of the resource manager providing the necessary I/O flexibility cannot be sufficiently stressed as the ability to multiplex channels and change I/O device dynamically is of great significance.

The other technique of importance is the use of an architectural model which caters for the necessary multiplicity. Capabilities for concurrent execution, introduced and supported by the architecture, ease the configuration of multi-channel interfaces. Without them, one must directly program the scheduling and interleaving unaided. Likewise, the target environment which provides the real communication devices and channels must provide the necessary flexibility and the mechanisms for synchronization (e.g. sound with video). The target operating system must contain the necessary facilities to allow for device multiplicity.

Representation Multiplicity

An interactive system has representation multiplicity if it offers the user multiple renderings (simultaneously or sequentially, on request) of one state element at any level of abstraction, or if it accepts multiple representations of the same input at the logical interaction, dialog, or functional levels of abstraction. In order to achieve this, presentation must be clearly separated from control and data processing during software refinement. This is supported by the use of an abstract architecture model in the design process. However, the systematic use of a good user interface design method may also help to identify the multiplicity that is appropriate for the intended users and adopted tasks.

I/O Re-use

The re-use of output from one interaction step as input to another step is an important convenience factor for users, similarly for re-use of previous input. Implementation of re-use is helped noticeably if the interface system adopts standards for inter-application communication. When design is done using a suitable architectural model, the possibilities for re-use will be established in the design phase and will considerably ease subsequent construction.

To a lesser degree, the adoption of common standards and conventions in other parts of the design may also help the designer to incorporate re-use of input/output data.

Human Role Multiplicity

Multi-user systems – and to some extent also single-user systems – must be designed to support users in different roles, being granted access to different sets of facilities. This is both a question of safety, where the system limits access for less privileged users, and of flexibility, where the system renders some information differently to different users. This calls for a design methodology that allows identification of the need for and the nature of this flexibility. It also calls for architectural models that can deliver what is required here. Since data should be used and presented in different contexts to different users, the consistent use of inter-application communication standards is essential. Also the underlying components – such as the database system and the I/O resource manager – must facilitate using the same data for different purposes.

Multi-Threading

Where a user interface system is required to provide the user with an opportunity to maintain several threads of activity at once, the correctness of the design is paramount. Correctness here must ensure that the interface system maintains separation and permits merging/splitting when called upon to do so, while maintaining the individual integrity of the functional cores. In this respect the need to use formal specification notations and tools cannot be over-stressed. The requirement here is for a process construct rather than just an interleaving construct/capability (as in production systems) where the actual threads of control are not easily isolated within a specific configuration. Once the (formal) specification is complete, however, the reification needed for the actual implementation can proceed in a variety of ways, for example, it may depend upon the architecture which may have been chosen for other reasons. If a choice of architecture remains, the selection should improve the multi-threading performance (by providing process facilities offering parallel execution of interface components) as well as guiding the appropriate re-ification of the specification during implementation.

It is again important to note that multi-threading of any kind necessarily involves interaction with any resource manager being used. Therefore, once again, care must be used in adopting the relevant protocols to achieve the desired end results. Note, for example, that many window managers can provide multi-threading between applications by letting users change their focus of attention between windows.

Non-Preemptiveness

Where a system is to be non-preemptive, great care has to be taken in its design to ensure that the correct desired temporal relationships exist be-

tween the user's actions and those of the system's interface. This is therefore primarily an issue at the session level of description, since it may place restrictions on the next user step in terms of intention and planning. The only way to ensure completely satisfactory specification of this is to use a formal technique which can be proven correct, for example using a dialog specification language, the use of which could be partially automated.

It is also important to realize that care must be taken in selecting appropriate protocols for use with any resource manager to be used with the interface design so that the specified relationships hold in the implementation. Without such a separate resource manager it must be noted that it is not possible to isolate the session level of description; the designer not using one would lose the ability to detect introduction of pre-emptiveness problems occurring at low levels of system abstraction.

Reachability

A system is said to be easily reachable if it allows users to navigate easily from any state to any other state. Reachability analysis of code, especially if the user interface modules are not well separated, is demanding and must be carried out using formal requirements to ensure a clean and correct system design, using formal specification languages and tools.

Layered specifications allow reachability to be established separately at one or more levels of abstraction. In practice most reachability analyses can be carried out most efficiently at the functional level. But, as further illustrated in Chapter 4, reachability is a pervasive property and therefore basically neutral to any architectural model.

Customizability

The term customizability includes **reconfigurability**, modifications to the user interface initiated by the user, and **adaptivity**, modifications initiated by the system. As already indicated when discussing relationships between several other properties, the architectural model used may have a prime influence on how easy an interface system may be to customize. The potential for customizability will also be enhanced by the use of flexible resource managers.

On the other hand, the adoption of inappropriate standards and conventions may adversely affect the possible customizability, particularly since standards – at least older standards – often prescribe one and only one way of doing something.

A customizable system must have some built-in flexibility, some possibilities left open. Provided that care is taken in parameterizing the design, the use of formal techniques supports well-defined user customizability. If care is not taken, the system could be difficult to customize.

Migratability

Migratability must be designed into the system at an early stage and is best achieved through use of a well-structured system architecture and a good design methodology. Architecture is relevant, as different forms of migratability exist at different levels of system abstraction, and an architecture that separates these levels will localize each form of migratability. Layered architectures support such localization, as do layered specification languages. The latter describe a system at different levels of abstraction. At the physical and logical levels of abstraction, input and output steps are described in detail using constructs appropriate to the interactive media that realize them. At the dialog level, these steps are treated as atomic events within some temporal structure. At the functional level, the system becomes an abstract data type, with abstract commands applied to and modifying abstract substructures.

Design methods should support identification of tasks that allow tasks to migrate between computer-supported ones and fully-automated ones. Generally, this will be at the functional level of interaction, but it can be even more abstract, at the level of user's goals, where many abstract commands may migrate to the system.

When goals and related tasks migrate to the system, some software *agent* becomes responsible for them. The same is true for lower levels of migration. At the logical interaction level, some agent must generate input events. At the dialog level, agents can follow scripts. At the functional level, agents can execute abstract commands. Whatever the mechanism, users must never find that commands work well in conjunction with other applications when they issue them but not when they migrate to an agent. In short, everything that works in the absence of migration should also work when it is added or invoked.

The ability to implement agents at different levels of system abstraction depends to an extent on user interface standards and inter-application communication standards. The latter may assist or obstruct the implementation of agents, generally by denying capabilities to programmers that are available to end-users (i.e. the system cannot do everything the user can). The former may impose patterns of interaction that obstruct optimal design of migratable functions, e.g. its visual design guidelines and display features may make it difficult to make the actions of agents salient (Clarke *et al.*, 1995).

3.5.2 Robustness Properties

The matching of robustness properties and software engineering techniques is summarized in Table 3.3. As illustrated by the first columns of the table, the design methodology, the use of an architectural model, and specification

languages and tools play important roles in the design of systems with good robustness properties. The influence on each property is discussed below.

Observability

The use of formal specification techniques in the design of an interactive system is a primary factor in preparing the system for observability, because the specification must contain all state elements of interest to the user. For example, the dialog level supports observability walkthroughs, which can isolate the values rendered at a particular interaction point. With some notations such analysis could be automated, and Chapter 5 addresses tools that do this. In order to assess observability it is important to use a design method that supports the assessment.

The adoption of a suitable architectural model will also throw light on how such observability may be achieved. Such a model isolates specific relationships between levels of description in user interface configurations. This is only possible in architectures that allocate different levels of description to different architectural components. Suitable architectures must further support explicit links between elements in different levels of description. For example, elements at the functional level of description could be linked to elements at the logical interaction level to expose the correspondence between a display item and the underlying value that it renders. There are many possible forms for such links, and they will affect the extent to which analysis of observability is supported.

Insistence

A system is insistent if feedback to the user is sustained and demands some user reaction. This is best provided for during design by using formal specification languages and tools, which permit the explicit specification of such sequencing requirements. Whether or not insistence can be obtained by an implementation depends, of course, to a large extent on the components of the target environment (e.g. absence of sound, no modal dialog boxes, no locking of display resources, limited graphical or text coding for emphasis). The designer, therefore, must consider such target environmental factors when planning for insistence.

Where insistence is achieved by some form of pre-emptiveness, the use of formal descriptions lets designers establish that the required pre-emptiveness has been achieved. Formal descriptions at the dialog level allow manual and automatic analysis of the persistence of information on specific interactive devices; and the design method must allow assessment of the property.

Table 3.3 *Relationships between robustness properties and software techniques. Entries indicate the relations:*
++ : is a primary mechanism to help achieve the property
+ : has a secondary effect in achieving the property
- : may have a negative effect on achieving the property
empty: the technique has no significant effect with regard to the property

	The Use of Software Techniques								
Robustness Property	Design methods	Architect. models	SW Re-use	QA Plann.	Specif. Lang.	I/O Res. Manag.	Target Envir.	UI Stand.	Comm. Stand.
Observability	+	+			++				
Insistence	+				++		++		
Honesty	++			+				++	
Predictability	++				++		-	+	
Access Control						++			+
Pace Tolerance	++						-		
Deviation Tolerance	+	++	+		++		-	+	

Honesty

No software engineering technique can guarantee that a system will be honest, neither misleading nor misinforming the user, because honesty is intimately linked to the users' perception of the system. In this context, however, a user's perception is frequently dictated by experience. Such experience is, like a standard, a distillation of what has been found to occur in the past. The adoption of appropriate standards, particularly in relation to the visible elements of the interaction, is therefore perhaps the best way of giving a user a subjective impression of 'honesty'. User interface standards can create, maintain and reinforce user's expectations on the use of interface controls, from the simple operation of pull-down menus to the standard contents of 'File' and 'Edit' menus (e.g. as specified in the Windows Application Development Guide (Microsoft, 1992)).

Allied to this adoption of such standards, however, the designer must not neglect the use of a design methodology with as much user involvement as possible to supplement and reinforce such subjective perception. Also, quality assurance procedures with user involvement further the construction of a system which users perceive as honest.

Architectural models that simplify the maintenance of observability will also contribute to the achievement of honesty. Similarly, anything that supports insistence will inevitably also support honesty.

Predictability

There are two key aspects of predictability: consistency of features and consistency of response time. The feature consistency aspect of predictability may be promoted by adoption of standards and common conventions in the same kind of way.

Desired temporal aspects of system behavior can only be achieved, if user involvement helps to establish actual parameters of appropriateness during the design phases. The designer must also be aware that hardware or software constraints in the target environment may spoil all attempts to achieve temporal predictability.

Both of these aspects of predictability effectively require that the interface system specification is logically complete – that a formal specification language has been used and that there are no unspecified behavioral components, so that every action/reaction is totally predictable. Architectural models which simplify the maintenance of honesty may – as a secondary effect – also support predictability.

Access Control

Access control may contribute significantly to the quality of multi-user systems with multiple human roles, but may also be of interest in single-user

systems, as discussed in Chapter 2. The basic problem of access control is to differentiate the users' access to inspect or alter certain parts of a system, according to the users' roles or tasks. The users may have different opportunities to view data, to execute task sequences, or to adapt the interface. Most of this is centered around the handling of input/output for data access, therefore the design must use appropriate I/O resource management facilities, and use of inter-application communication standards may also further the ease of consistent and planned access control.

Pace Tolerance

In order to construct a system perceived as pace tolerant by users, the design methodology must involve user tests, especially at late stages where a prototype is almost ready for field-testing. The designer must require that the target environment does not prevent pace tolerance (for example by the unavoidable presence of built-in hardware time-outs and undesired environmental software variations).

Deviation Tolerance

This is the capability of the system for backward or forward error recovery. It can be achieved only by using suitable abstractions and systematic refinements during design and construction. This means that an appropriate architectural model (which provides for 'undoing' support at different levels of system processing) and the use of formal specification tools are necessary.

Consideration should also be given to the adoption of relevant standards and conventions for error recovery. There is a wide repertoire of design features that support good deviation tolerance. If these are not features of a specific user interface standard, and this standard must be adhered to in a design, this may have a negative effect on deviation tolerance. Conversely, if the standard provides support for difficult and obscure features, the standard will have a positive effect.

The designer should not forget that error recovery is necessarily dependent upon being able to recreate some previous state. Undoing mechanisms make a major contribution to deviation tolerance. These can be difficult to implement in many current programming languages without some form of exception facility, although extensive support for rollback is provided in some database managers. The ability to provide undoing and error recovery is, therefore, inevitably limited by the state recording facilities (and their reliability) provided by the target environment and re-used software. This must therefore be chosen appropriately to satisfy the property requirements specified for the user interface system.

3.6 Conclusions

This chapter has addressed the internal software properties of a user interface system, properties that are additional to the external usability properties discussed in the previous chapter. While the internal properties are not directly visible to the users of an interactive system, they are still most important in determining the usability qualities of the system. Directly or indirectly they influence the external, more visible properties.

The three areas of software engineering techniques (methods, tools and standards) which have been discussed must be evaluated by the designer to establish that they do not impede satisfaction of the quality requirements laid upon the interactive system being designed. It is our strong conviction that the design and construction of a satisfying interactive system is promoted by the judicious use of properly selected tools and techniques. But the above discussion covers mainly *existing* software techniques, not all *possible* techniques.

Tables 3.2 and 3.3 reflect in some sense the current state-of-the-art in user interface development. A '+' doesn't mean that using that technique enforces or guarantees the property in question. The entries show where application of certain software techniques may help the designer to obtain quality in user interfaces, and the missing entries point to areas where more research is needed. The tables illustrate how important it is that the designer takes the target environment into consideration. Features and tools in the target environment may ease the achievement of several of the internal properties, but may on the other hand spoil some of the robustness properties.

The tables indicate the apparent current relative importance of each software technique. Three software techniques stand out: design methods, architectural models and specification languages. Each appears in around half of the rows in the combined tables. However, they differ in the balance between interactions as primary mechanisms and secondary effects. Specification languages are the most common primary mechanism. This is due to their role in establishing properties during design phases. However, they very rarely have any secondary effects, as their use does not pervade the development life cycle as other techniques do. In contrast, interactions with architectural models are evenly balanced between primary mechanisms and secondary effects. The effects of architectural decisions begin during the design phases and pervade all further development and (attempts at) maintenance and modification. As a result, architectural models interact with more properties than do specification languages. Design methods are equally ubiquitous, although they tend to be more secondary in their effects. When they are a primary mechanism, they operate like specification languages, establishing properties during design phases, but not pervading the development life cycle until system decommissioning.

Properties that hold for architectural models will generally be preserved by the actual software architecture of the installed system. The effect of architectural models is thus pervasive and less volatile. The effect of specification languages and design methods is less pervasive and more volatile, i.e. the benefits that they bring during the design phase are liable to evaporate, unless they can be preserved by other techniques. It is necessary to consider the role of various software techniques in preserving properties throughout the entire life cycle of the system. A repertoire of these software techniques under the heading of *tools and materials* provides the subject for Chapter 5.

Architectural models are considered first for two reasons. Firstly, their interactions with properties pervade and persist through software development. Secondly, they are *one* technique, and therefore the analysis will be more straightforward, being more amenable to an in-depth examination of some candidate architectures.

The aim of this book is to map out the space of interactions between the people that use software, the people who develop it, and the very software itself. The attempt at this is now largely completed. The topography has been surveyed, although some *terra incognita* remains (for example external properties associated with the principle of *learnability*.) The next three chapters move from topography to geology. Chapters 4 and 5 can be thought of as bore holes that will reveal the underlying structure of some regions of the space of interactions between properties and techniques. Chapter 6 attempts to validate the conclusions of Chapters 2 to 5 by analysing Air Traffic Control applications from the perspective of our properties and identified interactions with and between them.

Given the pioneering nature of this work, it is sensible to begin the next part of our survey with the most promising region. We thus now turn our detailed attention to software architectures for interactive systems.