CHAPTER 1

The Context of Interactive Systems Development

1.1 Introduction

Interactive computer systems are built in order to help people achieve some goals as efficiently as possible. Users at work have tasks to perform, and the systems they use should support these extensively and appropriately. This chapter establishes a context for discussing the *quality* of interactive systems.

Many different methods for the development of interactive software have been discussed in the literature. The purpose of this chapter is to outline the development scenarios chosen for discussion in this book. In order to discuss quality and quality goals for software development, agreement is needed on a basic vocabulary, an understanding of the development process, and a definition of the human roles that participate in the process. After defining the terminology used and the development phases considered, the chapter introduces human roles and outlines our vision of an ideal environment for development of interactive software.

Quality of a user interface shall be measured by measuring a number of properties of the interface and the computer system. Some of the properties are 'soft' and can only be defined and measured by taking the user's cognition and understanding into account; other properties are 'harder' and can be measured more easily by standard software engineering methods. The properties so defined can be ordered or grouped together in many ways, depending on which features are considered most important and relevant. We have chosen to distinguish two types of quality properties:

- From the user's perspective high quality means that the interface is pleasant, reliable, easily understandable, and that it has sufficient functionality, so that all the identified tasks can be performed with ease. These characteristics describe what we shall call *external quality*, and they will be defined in terms of a set of external, i.e. user-perceivable or at least inferrable, properties of the interface.
- From the software engineer's perspective, the user interface – or rather the software and hardware implementing this interface – is part of the system: the quality of the interface is judged in a way similar to other parts of the system. This kind of quality is called *internal quality*. It is

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

defined through a number of software and hardware properties of the system, such as modifiability, maintainability, and run time efficiency.

External quality is described using a set of task-related properties; internal quality is presented as a list of software-development-related properties. All properties are influenced by the design of the interactive system. Those properties that contribute to system quality must be considered explicitly during the development process: it is too late to think of them when the design has been completed. It is necessary, therefore, to discuss these properties in relation to software architecture and software development, including software engineering techniques and development tools.

The external properties will be introduced in Chapter 2, the internal properties in Chapter 3, using the context and terminology defined in this chapter. These properties are not completely independent (or mutually 'orthogonal'). Some may conflict in the sense that if a system is designed to have one property, another one may be very difficult to obtain. Others may automatically support each other, as demonstrated in Chapter 6 in the discussion of an example system.

User involvement in the entire development process is essential, because user requirements for the interface style – the look and feel of the system – vary from one user to the next, and also vary for the same user over time (as that person learns more about the system, or carries out different tasks). The same is true of functionality as the users understand it, i.e. the model that they form of a system's capabilities. These variations must be understood if a usable and useful system is to be produced.

The designer must also realize that properties are neither necessarily good or bad features. To make a good design is to make a proper selection among a number of choices. Design objectives can be seen as achieving a design which satisfies some properties (those required for achieving high quality in the particular design) and is free of others (those contributing negatively to quality).

Structuring interactive systems to support user goals requires a different set of skills than designing to meet functional requirements. Therefore this chapter concentrates on the development process for interactive systems, the human roles in this process, and the tool environments that can be used to support development. The aim in doing this is to establish a context for the discussion in the following chapters. A complete survey of structured software development is not attempted here, but we do define some generally used concepts and terms in relation to the construction of interactive systems.

1.2 Terminology

Subsequent chapters introduce and define a number of concepts that relate quality to software architecture. This section introduces some basic terminology used throughout the book.

1.2.1 Goal, Task and State

The user attempts to reach some goals using an interactive system to perform certain tasks. From the designer's viewpoint, the system may be perceived as a state transition machine that passes through a number of states during interaction with the user. The precise meanings of these terms as used throughout the book are as follows:

- a **goal** is a psychological variable, a state of the world desired by a person or a group of persons. A goal will not always correspond in an obvious or straightforward way to physical variables (e.g. eliminate unacceptable overruns, improve newsletter layout, make claims for research work more humble).
- a **task** is a procedure (a concrete action or set of actions) that is designed to lead to a goal from the current state of the world. Whereas goals are abstract, tasks are always rooted in the here and now. Execution of a task changes the current system state to a new system state, the *goal state*, which – hopefully – fulfills or corresponds to the goal as perceived by the user.
- an **interaction trace** is the particular execution of a task or a set of related tasks. It can usually be described as a sequence of steps (or a complex of interrelated but not necessarily sequential steps) of interaction between a user and a system, which takes the system from the here and now to the goal state.
  User steps can be described as articulations, i.e. purely physical actions, or they can include cognitive steps such as decisions and calculations. A trace extended with cognitive steps will be called a *Cognitive Task Description.*
- an **interaction point** is a significant, observable hiatus in an interaction trace.
- an **articulation** is a sequence of physical user actions that communicates a chosen command to the system. (Command is a concept at the functional level, see Section 1.2.2.)
- **task support** is any feature of an artefact or action of a person that supports task execution by directing users towards effective and efficient procedures, that is, task support enables users to achieve their goals directly and easily. Task support artefacts may be computerized, but may also be documents (manuals, guides, etc.).

- the **system state** or the internal state is the set of values within a system that affect its present and future behavior. It is represented by a vector of all variables in the system, where each variable is a state element.
- the **observable state** is the observable part of the system state, i.e. those system data to which a user *may* obtain access (but they need not be presented at once).
- a **rendering** is a sequence – or complex – of physical system actions that communicates some observable state elements to the user.

In our terminology a goal is a state (of the world or of the system), while a task is a method or procedure that can be executed in order to reach a new, desired state. A task execution may be wholly manual (where the user effects the task execution), wholly automated (where the user just monitors the execution), or partially automated where the user's role varies from being an obedient source of information to being the manager of operations. The focus in this book is on partial automation of task execution, within the extremes of manual tasks and automation, although we occasionally address monitoring of automated processes. The overall concern in the following chapters is how to construct good computer-based task support.

1.2.2 Levels of Abstraction

Design becomes more abstract as attention moves away from communication channels, and the encoding of information on them, to the conceptual structure of work domains. Designers must work on several levels of abstraction, and each level brings its own concerns and knowledge sources. Such levels are well established in the HCI literature. For example, with the Command Language Grammar (CLG) Moran (1981) introduces seven layers of refinement used to structure the design process, and the GOMS method (Goals, Operations, Methods, Selection rules) by Card *et al.* (1983) introduces four different layers for task modeling, which are similar to, but not identical with, our levels as described below.

We distinguish between four levels for interchanges with an interactive system, where each level is a refinement of the earlier one. At each level of abstraction some data objects and operations are described as are event sequences; by an *event* we mean a 'unit of action', i.e. some data transfer and some process execution which, at this level, is perceived as one step. At a lower level of abstraction, each event usually becomes a sequence of lower level events – or a set of not necessarily sequentially executed events. The four levels of abstraction are:

Functional level – the highest level of abstraction within the system. At this level the operations (or abstract commands) and objects provided by the system are described. It is the first level below the 'task level'

which we consider as being outside (above) the interactive system. The term *command* is used here in the same way as in the PIE model (Dix *et al.*, 1993) to denote a single user action at this level of abstraction.

Examples: Three examples of (unrelated) functional level events or abstract commands are:
(a) start Draw program.
(b) set date and time.
(c) convert Celsius temperatures to Fahrenheit.

Dialog level – the level concerned with the temporal behavior and the interdependencies among the operations and objects. (This level is sometimes called the 'session level'.)

Examples: At this level the three functional level events from above are described in a little more detail:
(a) open DrawImage window.
(b) select month; advance month; select date; ...
(c) enter Celsius temperature; show Fahrenheit temperature.

Logical interaction level – the level of 'how to do the interaction' with some generalization over lower-level events and with reference to presentation entities rather than raw device values.

Examples: At this level dialog events like 'open' and 'select' are split into logical events on presentation entities:
(a) move mouse to DrawImage icon; click mouse.
(b) move mouse to menu; move mouse to 'month' item;
click mouse; ...
(c) type Celsius value in input field;
show Fahrenheit value in result data box.

If the system accepts spoken input the first example could be:
(a') say 'Open DrawImage'.

Physical interaction level – the lowest level of abstraction describing 'what really happens during interaction'. The description needs no references to display state or system state. Some call this the 'keystroke level', but others mean the logical interaction level when they say keystroke level.

The description of the example (c) above now 'explodes' into:
(c) move mouse to position (450,780);
button down at Fri Oct 22 14:18:36.260 BST 1994;
button up at Fri Oct 22 14:18:36.350 BST 1994;
type the actual sign, 2 digits and <Return>;
display the resulting sign and digits in data box at
position (650,780).

At the lowest levels (physical and logical), the designer can draw on perceptual and motor psychology. The distinction between the logical and

the physical level is sometimes very subtle, and it is not always needed because the underlying system may automatically take care of all the details at the physical interaction level; the designer needs only to specify the logical interaction.

The dialog level design is more concerned with end-user planning and activity structures. Design at the functional level deals with the conceptual objects and the abstract commands users may perform on those objects in order to perform *tasks* to achieve *goals*.

The users' goals and tasks are considered as being something outside the interactive system itself, but the interactive system is an implementation of the objects and operations intended to help the users to achieve their goals.

The levels of abstraction introduced above will be reflected in the architectural model of an interactive system discussed in Chapter 4, where a functional partitioning is introduced in close relation to the levels of abstraction.

1.3 The Development Process

The development process for all systems (whether interactive or not, computer-based or not) is usually considered as a phase structure which distinguishes logically separate activities. Development models integrate a collection of methods that support different phases. Different models have slightly different phases, but most identify the following to a greater or lesser extent:

- identifying the idea or problem in a given domain (Problem Analysis);
- determining requirements (Requirements Specification);
- outlining the system design (System Design);
- designing the software structure of the system (Global Software Design);
- detailing the design (Module Design);
- constructing modules (Coding or Module Construction);
- testing modules (Module Test);
- integrating and testing the system (Integration Test);
- testing the finished system against the system design (System Test);
- installing and testing the final system versus the requirements (System Acceptance);
- maintenance (sometimes called sustaining engineering).

During *Problem Analysis*, the need for a system, the nature of the domain in which it will operate, and the needs of its users and other stakeholders are examined. The result of the phase is a statement of the problem to be solved by the system, in the language of the users.

During *Requirements Specification*, or requirements capture, constraints are identified and acceptance tests may be specified. An important subphase is *Requirements Analysis*, where a user's problem formulation is analysed and transformed into specifications. Requirements often take the form of logical constraints on abstract models of possible final systems. In some developments, many of the high-level features of the final design are decided upon during this phase. The user interface could be specified during this phase. But often user interface specification is delegated to programmers during construction phases. This explains many of the problems end-users have with interactive systems. It is already important at this stage to introduce quality goals for the project which guide formation of quality plans for subsequent development phases, i.e. methods for achieving the quality goals.

The result of the phase is a description of the functionality of the system, constraints in its environment and quality goals. These specifications must be approved by users or customers.

During *System Design*, possible ways of transforming requirements into solutions are identified. Solutions are expressed as (abstract) models of the final system. One model, the physical architectural model, decomposes the system into *modules*. The result is the external specification of the system, i.e. a specification of a solution as perceived by the user; the solution must meet the requirements from the previous phase. The System Design document also includes a plan for the system test.

During *Software Design* (also called Global Software Design) the global software architecture is chosen, and the main components of the system and their interfaces are specified. The result is a software design document describing this global structure of the system's software.

During *Module Design*, modules are progressively refined until the major software structure of a system has been detailed. The result is the detailed description of all software modules and their interrelationships.

During *Coding* (or Module Construction) modules are implemented and debugged to provide the result of this phase.

During *Module Test* and *Integration Test*, the implemented modules are progressively integrated and tested until the final system is assembled. Each integration step is accompanied by a collection of tests.

During *System Test* the system is tested to determine whether it meets the external specifications as set up in the System Design document. This test is guided by the test plan (acceptance test) set up during system design.

During *System Acceptance*, the system must pass the acceptance tests specified in the requirements. The final system is used 'for real' by end-users. The system may go live in a series of steps. This is the last development phase.

During *Maintenance*, the system is exposed to regression testing after each (code) change. A regression test consists of performing all those de-

velopment tests (i.e. module and integration tests) that were used during the development of the now altered components. This is done to ensure the correctness of the modification.

Development of new systems is costly and time-consuming; it is important to re-use existing tested components, whenever possible. This holds for the entire development process and for modules at all levels up to complete designs. Therefore, it is important to introduce the concept of a quality plan into the development process. The *Quality Plan* defines quality goals and indicates methods and tools which may be used to reach the quality goals. The plan stretches across the development phases; test methods must be used to check on the achievement of quality goals, as discussed below.

The above sketch of the development process is by no means complete but has been found sufficient for the analysis in subsequent chapters. Clearly, a full description of software development needs much more detail, as found in the literature on software engineering in general.

1.3.1 Development Models

There are several different development models, or arrangements of the phases, currently in use. The effective management and control of the process is related to the model selected. One popular model, the *Waterfall*, connects phases into a pipeline: all prerequisite work for each phase is undertaken before that phase starts. This model emphasizes cost estimation and control. It mitigates some risks by ensuring that work is undertaken in a realistic order. Effective risk management is essential because there is generally an element of research or new work in all computer system development.

An alternative structure, the *V-model*, relates each development phase not only to its immediate predecessor and successor, but also to the construction and testing phase on the same level of detail. Requirements Specification deals with the usage of the total system, as does the System Acceptance. Acceptance tests are created as part of the specification and used during the final installation. Software Design (where a system is decomposed into modules) is arranged at the same level as Integration Test (where modules are combined to check the correct interplay between the modules). Module Design is on a level with Module Test, because modules are tested against specifications from the module design. The V-model is illustrated in Figure 1.1. Only development and test phases are considered parts of the V; thus Problem Analysis and Use and Maintenance phases are kept separate (and shown just above the V).

This simple V model is still a Waterfall model in that it does not allow backtracking to a phase once development has advanced beyond it. This simplifies management, but may compromise quality, a risk that may be

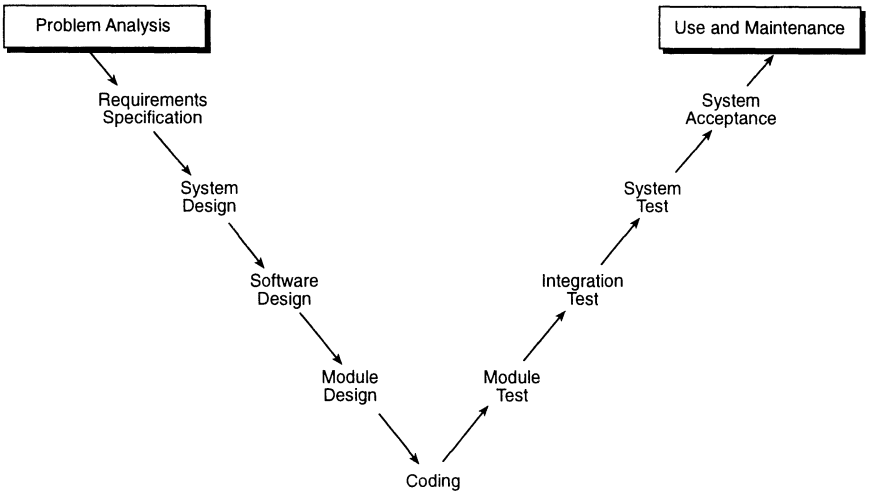

Figure 1.1 *The simple V model for software development. The arrows show the temporal order of the phases.*

greater than the ones that a Waterfall approach does avoid. Decisions made before Module Design inevitably rely on assumptions that could not be thoroughly validated when they were made. This is not only a question of time and resources; it is impossible to predict the impact of design decisions. What looks acceptable on paper may turn out to be incomplete or too specific, or to have other unacceptable consequences in practice. During the design process it is also very common to use implicit assumptions, since there is rarely time for a complete and rigorous problem analysis. A further problem with Waterfall structures is that even explicit assumptions that were thoroughly validated during Requirements Specification and System Design may later become invalid. The world changes.

Solutions to the limitations of waterfall structures, where all project steps are carried out as single steps in a forward sequence, exploit iteration in development by allowing steps several phases forward or backward. In the resulting iterative process, a project may cycle through the same phase several times. It has become popular to characterize this type of development process as a spiraling, iterative cycling through the phases (Boehm, 1988). We prefer to consider iteration as an extension to the phase sequence of the simple V model. Figure 1.2 shows possible iterative steps in a *V-model with backtracking*, where each design phase is still matched by a test phase.

Each backtracking step results in some 'recovery' that extends, corrects or refines existing inadequacies in previous phases. Thus during System Design, gaps, errors and unclear definitions in Requirements Specifications may be detected. During Module Design, problems with the global software

decomposition may be detected. During Coding, problems with the module refinement may become apparent.

During Module Test, problems with the module refinement, stubs, drivers and system decomposition may become apparent, as may problems with the system level decomposition and the requirements specifications. The problems that emerge during System Acceptance are often more subtle errors that only show up in the full-scale system where all components interact. Such errors are typically due to very early decisions during Requirements Specification (and perhaps System Design), as problems with other phases are generally detected during System Test.

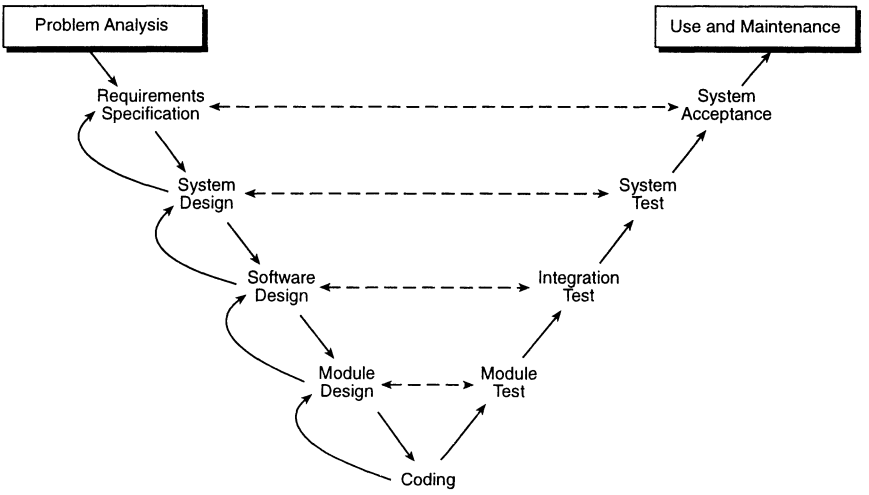

Figure 1.2 *The V model with backtracking. The solid arrows show the usual temporal order of the phases and the backtracking steps. (Backtracking may also jump more than one phase back.) The dashed arrows indicate that test plans must be made for each development phase, and the testing results give feedback to the design phases.*

An alternative type of model is the *evolutionary prototype*, where the phases of the V are undertaken one function at a time, as suggested by Bersoff and Davis (1991). Functions and features in a high-level design are given a priority in order of importance or of anticipated stability. The functions with highest priority are then developed to completion first. The development cycle is repeated for the next most important functions, etc.

Backtracking steps give rise to iteration as previous phases must be revisited for remedial action. However, there are other forms of iteration. *Speculative* steps are also possible. Here analysis, specification and systematic design may be skipped in order to test out ideas and hypotheses by constructing prototypes. This *rapid prototyping* results in throw-away

prototypes, corresponding to quick runs through all phases in the V. Once ideas have been tested and hypotheses confirmed or rejected, this information is fed back into the requirements phase, and the design proceeds until new uncertainties are encountered. At such points, development may be suspended while another rapid prototype is constructed to address the uncertainties.

Innovation, with all its uncertainties, requires some form of prototyping. One way of trying to 'foresee the unforeseen' is called *participative development* where users are involved in the design phases (the left-hand branch of the V), see Muller *et al.* (1993). During participative development only some aspects of the needs of some individual users may be assessed and hopefully fulfilled. On the other hand, a prescriptive approach, which assumes users are wholly predictable, is neither practical nor appropriate. This perspective is an important context for the development of properties in Chapter 2, since it restricts our approach to the analysis of tasks and their idealized execution. We avoid properties that rely on some model of the user, but stress the need for involving real users in testing an interactive system. Indeed, some of the properties can only be tested for (and perhaps measured) through observations of users' interaction with the system.

The models described above do not incorporate explicit steps to re-use parts of other systems. Still, not everything about a system is new. If something has been done before, if its applicability is well understood and if it has been implemented in a re-usable form, then it should be re-used. Thus during Requirements Specification and/or System Design, existing code that supports a required function should be identified, and the system should be designed to use this code. Thus development may not begin with a clean slate.

1.3.2 Interaction Design and the Development Process

The above account of development is applicable, with modification, to batch systems, to embedded systems and to systems that interact with human end-users. Only the latter are the subject of this book.

Research into Human–Computer Interaction (HCI) and into Interactive Systems Design has added specialized techniques and outputs to each phase of the development process outlined above. HCI approaches:

- model new aspects for system design by introducing task, performance and conceptual models (the latter describe systems at the functional level);

- introduce new detailed design concerns related to output formatting, interaction technique, and the use of color and sound as well as other media and modalities in information coding;

- add new software components especially for the dialog, such as help, history, undoing, macros, tailoring, tutoring;
- produce new development models with different orderings of development phases, e.g. designing the user interface first;
- create new forms of testing, e.g. formative and summative usability testing;
- give rise to new forms of installation plans, e.g. special training plans for dialog-intensive systems;
- introduce new problems of maintenance, e.g. for self-adaptive systems that change the dialog by exploiting emerging users' pattern of usage.

Many of these new activities concentrate on the design, development and revision of the perceivable user interface to the system. Users interact via *communication devices* such as speech input or output, graphic displays and haptic devices (mice, tablets, etc.). A communication device is thus anything which transfers coded information between the user and the computer. Designers must pay careful attention to the selection of these communication devices and the manner in which they are used.

These communication devices are mostly concerned with the lowest level of abstraction, the physical interaction level. However, interaction design is much more subtle and complex than designing communication devices. Users will attempt to make sense of the underlying temporal and conceptual patterns of interaction, so designers must specify these explicitly and be confident of their adequacy at each level of abstraction.

Schematically, the design process starts with a task analysis identifying the tasks to be supported. At the functional level, the task steps are conceptualized as abstract commands applied to objects. These are then refined through the remaining levels into specific sequences of renderings and communication devices at the physical interaction level.

But the quality of the user interface is dependent on features – and combination of features – from all these levels, as will be evident from the exposition in Chapters 2 and 3. Therefore the developer of an interactive system must include quality aspects from the very beginning of a design process.

Design of interactive systems requires continuous capture of requirements, constraints, and modifications throughout the development process, certainly up to the completion of the design phases. Thus, designers require iterative development and backtracking transitions, although the latter can be reduced by initial speculative prototyping. These approaches recognize that many non-functional requirements cannot be specified in advance of the construction and demonstration of possible solutions. However, iteration must be constrained if diminishing returns are to be avoided: it is often said that the first 20% of any effort produces 80% of actual improvements.

The synthesis of a development process for interactive systems requires answers to four key questions:

1. Where/how does the user interface get designed and developed?
2. How are users involved in the process of design?
3. What are the relationships between user interface development and the remainder of the development process?
4. What are the relationships between user interface management software and the remainder of the interactive system?

The relationship between user interface management software and the remainder of an interactive system is one of the major foci of this book. It draws on the large body of research on user interface software technology. This research addresses mainly internal software properties, i.e. properties not directly perceivable by users. The research is concentrated on developing conceptual, logical and physical architectures for the software of interactive systems. An important issue in designing interactive systems is keeping the software components for user interface functions separate from those of the rest of the interactive system, which we call the *functional core.* The functional core provides the computational realization of the problem domain functionality for an interactive system. User interface components represent this functionality to end-users and support them in the use of these representations.

1.4 The Development Process: Human Roles

The development cycle as described above gives rise to a number of roles which may be filled by one or more humans; conversely, a single human may participate in one or more roles, and very often each person in a development team performs several of the roles discussed in this section.

This section presents an analysis of the interactive system development process based on the human roles within it. Each role is associated with a subset of objectives that arise during the development of an interactive system. The division of labour is compatible with a basic assumption of software separability into user interface and functional core components. Roles may thus be specific to design at a particular level of abstraction. Some of the roles listed below are outside the scope of this book, as they do not participate directly in the software development process. They are mentioned here for the sake of completeness.

All role objectives are described informally in this section. However, some will be given a more precise content in Chapter 2, which provides a catalog of general interactive properties. We view a *property* as some aspect of the software quality of an interactive system, and several of the properties may be taken into consideration and determined at one or more levels of abstraction in a design. Each property represents a standard by which an

interactive system can be evaluated, and an important part of the design process is to ensure that the system under construction has some of the properties (those desired for this particular system) and does *not* have certain other properties.

1.4.1 Human Roles

Each human role has a set of tasks to perform. For each task there are constraints on the starting point for that task and the quality of the output from the task, i.e. the development objective is associated with each role. Most tasks are complex, and quality is difficult to attain without *task support*, so for each role the task support requirements are also outlined.

Client. The client assesses the intended scope of the project, and provides payment for the resources to design and implement an interactive system. This role needs task support for outlining – at a high level – requirements, acceptance tests, training plans and installation schedules.

Project Manager. The project manager is responsible for making available the resources necessary to complete the entire design and implementation, and for scheduling the resources for near-optimum usage. This role needs task support for general software engineering tasks such as cost estimation and control, task scheduling, and life-cycle management.

User Representative. The user representatives are the problem-domain experts who have knowledge of the application domain. They provide feedback at as many design phases as possible and participate in usability testing of prototypes and final systems. The user representatives should represent as wide a range of potential end-users as is practical. Their key objective is to propose and validate requirements and their interpretation as embodied in the software. They need to be provided with early prototypes and with tools to assist in evaluating the prototypes.

These first three roles are important for defining the framework for the development but are not discussed further, because they do not participate directly in the software development work. All the roles discussed below are directly involved in the development or the use of the interactive system.

Requirements Specialist. Requirements specialists perform needs and task analysis to determine potential end-user requirements and tasks, and to explore end-users' conceptual models of the work domain.

This role can use task support for requirements elicitation, data collection, cross-referencing, video capture, repertory grid analysis, user profiling, organizational profiling (business goals, privacy, security, safety), technical profiling ('sizing' hardware, performance), requirements animation, scenario generation, task description and analysis, and user interface style selection/specification.

The next roles deal with the design and implementation phases. As mentioned above a system may be considered as consisting of a functional core part and a user interface part. This division is reflected in the roles described below (although the V model does not explicitly show that partition). The designer and the implementer may often be the same person (also called system engineer),who may, at least for smaller systems, design and implement all parts of the systems.

System Designer. This role may be split into three sub-roles:

(i) The *Interactive System Designer* is responsible for the initial system level design. Once the system level design is complete, the System Implementer is responsible for managing and coordinating the activities of the User Interface Designer, User Interface Implementer and Functional Core Designer/Implementer. The Interactive System Designer also pays attention to the choice of implementation platform, development costs, and concurrency and synchronization issues that arise from distributed software components. The operating system, development tools such as user interface toolkits, existing implementations of the functional core, memory size, speed of processor(s), and the interface devices available may all impose constraints on the design and implementation of the interactive system.

A key task for the role is the identification of the state vector at the highest level of abstraction, the conceptual structure of the application domain. Another task of the Interactive System Designer/Implementer is to ensure that the system and its components possess the desired properties, such as re-usability, modifiability or reconfigurability (discussed in the following chapters).

(ii) The *User Interface Designer* specifies the more detailed dialog design. This requires expertise in ergonomic principles and/or aesthetic sensitivities for dialogs, for user support (help, history, etc.), and for encoding via communication devices to create a coherent, concrete representation of data for end-users. The designer must ensure the usability by aiming for the external properties discussed in the next chapter, subject to the constraints given by the actual application domain and the requirements. Good usability is often accomplished by prototyping user interface fragments and evaluating the end-users' interactions with those prototypes.

(iii) The *Functional Core Designer* is responsible for the logical decomposition of the non-user-interface code (the functional core) and for selecting existing tools, libraries (databases, numerical packages) and designs.

Sub-role (i) needs task support for conceptual model design, task allocation, hardware selection and transformations from requirements to specifications.

Sub-role (ii) needs efficient and effective access to a collection of user interface components (e.g. a library of interactor classes), and further task support for presentation design (e.g. bitmap editor, icon editor, and layout editor), interactor design, animation, and dialog design. Where task support takes the form of computer tools, ease of use should be established for non-programmers. It should be possible to use individual tools in isolation.

Sub-role (iii) needs task support for the Interactive System Designer role plus tools for specification of exported objects, binding services, and access control.

Implementer. Like the designer role, this role may be split into three sub-roles:

(i) The *Interactive System Implementer* is responsible for managing and coordinating the implementation activities and needs task support for data dictionary use, version control, re-use of components and configuration control.

(ii) The *User Interface Implementer* applies expertise in conceptual, logical and physical software architecture design, and software specification to generate formal descriptions of the user interface. The Implementer also applies programming skills to develop a working user interface. The role needs task support for system modeling, specification, compilation, verification, validation, debugging, step-through/animation as well as style realization.

(iii) The *Functional Core Implementer* implements the specified objects that belong to the functional core of the system. The role needs task support for debugging, regression testing, validation and data dictionary maintenance in addition to the support required by the User Interface Implementer.

Validator. Throughout the development and testing process it is important to focus on quality and validation. This role has the responsibility – in all phases – to ensure that the objectives of a quality plan are achieved. This role may also be split into sub-roles:

(i) The *Quality Specialist* sets up a quality plan for the entire development project and manages the testing when it is carried out. This requires task support for project management (much like the Project Manager), verification, validation and quality assurance tools.

(ii) The *Usability Specialist* applies knowledge in experimental and cognitive psychology to design, implement and conduct usability evaluations (user testing). The purpose is to determine ease of learning and use, paying attention to usability measures such as time, error rates, correspondence between goals and tasks, and subjective satisfaction. This role needs task support for evaluation (contextual evaluation criteria), experimental design (scenarios, user selection), test management, data

gathering (video, multi-level logging) and data analysis (protocol analysis).

(iii) The *Software Validator*, designs, implements and conducts tests to determine the completeness, adequacy and robustness of user interface software. This role needs task support for rehearsal, evaluation, and testing (e.g. simulation, playback, check lists).

The two last roles (User and System Administrator) concern the final system. Like the first two roles (Client and Project Manager), the last two do not participate in the software development. But they are important as representing the persons using the final system.

User. By users we mean end-users of the final system, the persons solving their tasks helped by the interactive system.

System Administrator. The system administrator keeps the interactive system running and controls access to computational resources and files. The administrator also receives error reports and initiates maintenance when needed. This role needs task support for configuration management, version control, access control, resource allocation, database administration, bug tracking, and maintenance control.

1.4.2 Human Roles in the V Model

The main part of the system specification, development, and testing are performed by the roles requirements specialist, system designer, implementer, and validator with their sub-roles. There is a simple relation between these roles and the phases of the V model. The phases on the left side of the V model (problem analysis, specification, and design) are performed by requirements specialists and system designers. The bottom phases (module design and coding) are performed by implementers, while validators cover most of the right side of the V model.

In each role, an individual works with some material that is transformed into some other material or product (or is related to it) by means of some tools. By *material* we mean any document, data collection, or program which is part of the system under development. In the development process the individuals may use *tools* on some materials to generate new materials. In Chapter 5 we shall take a closer look at tools and materials and how they may influence the development process and the quality of the final system.

1.5 Interactive Software Development Environments

The problems considered in HCI research are relevant, not only to the development of specific end-user applications, but also to the development of tools for constructing such interactive systems. Support software both for

the development of user interfaces and for the management of interactions is essential. Each development role is associated with a set of objectives. Software support for the satisfaction of these objectives is both feasible and desirable. In this section, we outline a comprehensive support environment for the roles described in the previous section.

Software engineering environments – also called software development environments or computer-aided software engineering (CASE) tools – aim at making program development and construction efficient, without loss of functionality. Here we focus on properties of the *Interactive Software Development Environment* (ISDE), i.e. those parts of programming environments that are directed specifically toward the efficient construction of interactive systems. An ISDE is a general, comprehensive environment that provides support for a wide range of development roles.

The term UIMS, User Interface Management System, was coined in an attempt to promote the concept of separating the interface part of a system from the functional core (the application part). Complete separation is not possible in any but the simplest systems, and the development of all parts must go hand in hand. Therefore, a somewhat broader view of the development environment is taken here, where the interplay between interface part and functional core is taken into consideration. The term UIMS is not used below, but it would correspond to *some* of our User Interface Development Environment, *some* of our Binding Services, and *some* of the resulting interactive system in Figures 1.3 and 1.4.

General and comprehensive support environments for interactive systems development are possible. But the construction of ISDEs is complicated by the regular arrival of new communication devices for user interface implementation: (glass) teletypewriters were superseded first by cursor addressable text displays, and then by raster graphical displays, which in turn have been supplemented with mice, touch screens and audio input/output devices.

Although the functionality offered by interactive systems varies from one system to the next, much of the software processing in flexible, effective user interfaces is largely separable from the intended functionality. The same run time support code can manage the user interface for different functional cores. The code needed depends on the machine architecture, the operating system, the communication devices being used, and user preferences for interfaces – much more than on the purpose/function of the system. Also, the variety of communication devices does not imply completely disjoint design rules for constructing user interfaces. Many choices apply regardless of the communication device, so similar tools can facilitate development.

Many attempts have been made to provide practical tools that assist with the development of user interfaces, and with management of the interaction between the user interface and the functional core. Typically, the support provided varies with: the hardware being used; the operating system being

used; the preferred look and feel; the assumptions of the user interface tool designer about what sorts of interaction may be required; and assumptions about how interactive system designers work. Below, we abstract from the variability of available tools and accommodate them within a general tool framework.

An ideal environment would provide designers with a single source of support that accommodates differences between operating systems, hardware, communications devices, and interaction modalities and styles, considerably easing the task of porting among different technologies. In turn, descriptions of such an environment can support designers who must develop such software support.

The main task in developing interactive systems is specifying their rendering via communication devices, their surface behavior, and their underlying functionality. There are three basic requirements for such specification tasks. An ISDE should allow:

- simple specifications of simple systems;
- declarative specifications of non-procedural aspects of interactive systems;
- interactive specifications of procedural aspects of interactive systems.

Such an environment contains tools that support the complete development of interactive software, and thus support all phases of development (as outlined above). Figure 1.3 shows the general structure of an ISDE.

An interactive system is developed on the basis of requirements by using an ISDE. The ISDE sets up suitable *Environments* for the different development tasks and offers the designer and the implementer a number of *Services*. At the coarsest level of task description, ISDE Services are used to create more detailed design specifications of the interactive system. The more specific tasks of developing a functional core and a user interface are supported by two other components: a *User Interface Development Environment*, UIDE; and a *Functional Core Development Environment*, FCDE. The results of using UIDE and FCDE are a number of software modules, and this output of the two subsidiary environments is combined into an instantiation of an *Interactive System* using the *Binding Services*. Although only one interactive system is shown in Figure 1.3, an ISDE should be able to support concurrent multiple instances of interactive systems, as well as several functional cores and several user interfaces.

The ISDE Services must support the initial specification phase, the project administration, and the later validation and evaluation of the interactive system. Hence they must contain: (i) general specification tools; (ii) tools for project management; and (iii) tools for testing of complete interactive systems.

Examples of specification support are tools for conceptual model design, coarse-grained task analysis and cognitive modeling.

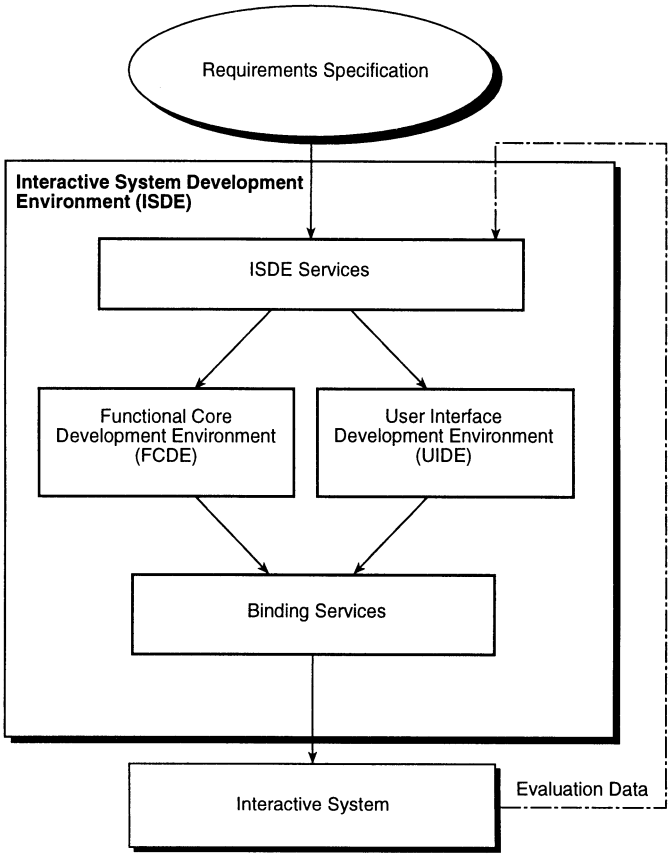

Figure 1.3 *The major components of an ISDE.*

General project administration is supported by tools for version control, configuration management, archiving and re-use, and system testing.

Support for testing comprises tools for: rehearsal, simulation and playback; video and software logging/analysis.

Binding Services contain tools with mechanisms for instantiating the interactive system from specifications and modules produced by subsidiary environments and for linking code instances within the final system. The latter functionality may exploit dynamic binding (of components created within a session) or static binding (of ready-to-use components).

Figure 1.4 gives a more detailed view of an ISDE, the subsidiary environments UIDE and FCDE, and their tools. The figure also shows which human roles the different parts support. The tools and the related materials are discussed further in Chapter 5.

The main input to FCDE and UIDE is an interactive system specification

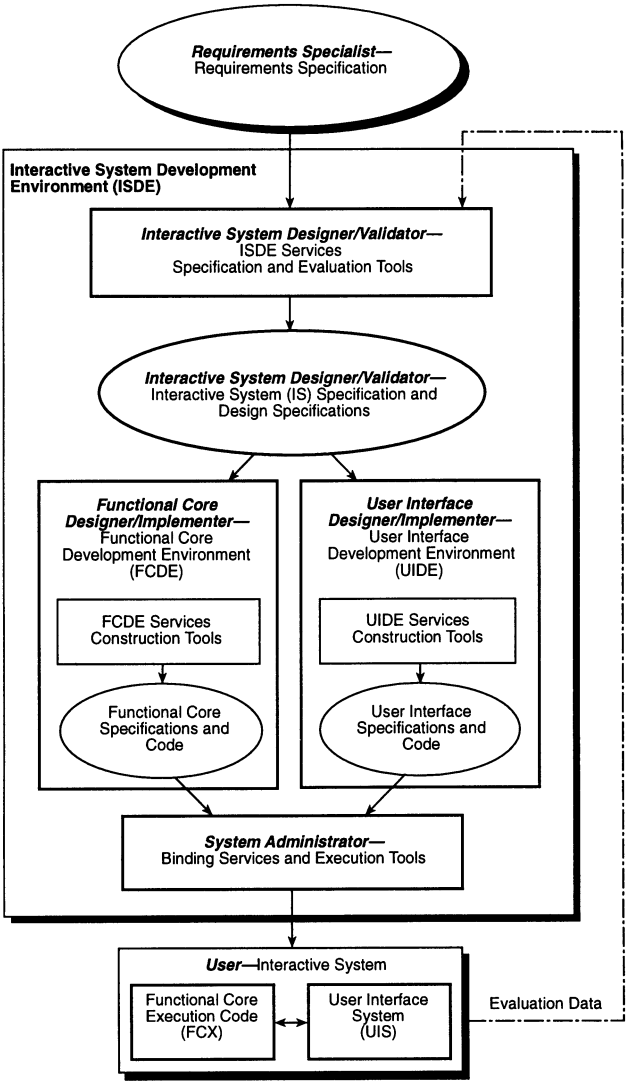

Figure 1.4 *The components of an ISDE and their use by the human roles. Each human role is shown in bold face within the components supporting this role.*

produced with the aid of ISDE-Services. The User Interface Development Environment (UIDE) supports the iterative development of user interfaces. It offers services – *UIDE Services* – by means of which the designer creates user interface specifications and code (UI modules).

The UIDE Services are a collection of construction tools such as presentation design tools (e.g. bitmap editor, layout editor), finer-grained task

analysis and cognitive modeling tools together with user system protocol design tools (e.g. dialog control editor).

The FCDE, the subsidiary development environment for domain functionality, may be decomposed similarly. The FCDE Services are a collection of construction tools by means of which the software implementer constructs a specification of the functional core of the system and corresponding code modules.

The Interactive System consists of two components: the Functional Core (abbreviated FCX because it is the executing code), and the User Interface part (UIS). This reflects the division of labour between the subsidiary environments, in that there are run time components for the UIS and the FCX. This division is somewhat conservative, as there are experimental approaches to *virtual separation* that support separation in the design environment, but do not preserve this at run time (Shevlin and Neelamkavil, 1991). In this case, the Binding Services restructure the run time architecture around common patterns of interaction, in much the same way as optimizing compilers restructure generated code.

This completes the brief overview of ISDEs, their components and their internal data flows. From the division of labour as discussed here, we can identify some requirements that ISDEs should fulfill:

- provide means for configuring high-level components, rather than just low-level ones like the widgets offered by a toolkit;
- provide a set of link classes for combining components;
- generate components with well-defined functional roles and interactions with system properties;
- provide clear rules for restrictions on component interrelations and provide means for enforcing such restrictions.

Thus, ISDEs should support component configuration at all four levels of abstraction, and be clear about software architectures that can be formed within them. Chapter 5 discusses this in more detail, and shows other ways of putting components together to form an interactive system.

1.6 Summary

We see software development as a structured process with well-defined human roles. Both the process and the roles have to be extended and specialized when the system is interactive. Such extensions further complicate an already complicated milieu. It is therefore essential that software tool support be provided for each human role in the development of interactive systems. The main design goal for each tool is to maximize the quality of the final interactive system, given the available resources. Such quality must be defined either as properties of the final system, or as properties

of end-user interaction with the final system. The next chapters begin by analysing general properties for interactive software and then look in depth at architectural models which can guide the construction of interactive software. Once all these structures have been adequately described, Chapter 5 discusses the pragmatics of ISDE design, by addressing the tools and materials with and from which they can be formed.