

Design Principles for Interactive Software

EDITED BY

Christian Gram and Gilbert Cockton

SPRINGER-SCIENCE+
BUSINESS MEDIA, B.V.



IFIP

Design Principles for Interactive Software

JOIN US ON THE INTERNET VIA WWW, GOPHER, FTP OR EMAIL:

WWW: <http://www.thomson.com>

GOPHER: <gopher.thomson.com>

FTP: <ftp.thomson.com>

EMAIL: findit@kiosk.thomson.com

A service of **ITP**

IFIP– The International Federation for Information Processing

IFIP was founded in 1960 under the auspices of UNESCO, following the First World Computer Congress held in Paris the previous year. An umbrella organization for societies working in information processing, IFIP's aim is two-fold: to support information processing within its member countries and to encourage technology transfer to developing nations. As its mission statement clearly states,

IFIP's mission is to be the leading, truly international, apolitical organization which encourages and assists in the development, exploitation and application of information technology for the benefit of all people.

IFIP is a non-profitmaking organization, run almost solely by 2500 volunteers. It operates through a number of technical committees, which organize events and publications. IFIP's events range from an international congress to local seminars, but the most important are:

- the IFIP World Computer Congress, held every second year;
- open conferences;
- working conferences.

The flagship event is the IFIP World Computer Congress, at which both invited and contributed papers are presented. Contributed papers are rigorously refereed and the rejection rate is high.

As with the Congress, participation in the open conferences is open to all and papers may be invited or submitted. Again, submitted papers are stringently refereed.

The working conferences are structured differently. They are usually run by a working group and attendance is small and by invitation only. Their purpose is to create an atmosphere conducive to innovation and development. Refereeing is less rigorous and papers are subjected to extensive group discussion.

Publications arising from IFIP events vary. The papers presented at the IFIP World Computer Congress and at open conferences are published as conference proceedings, while the results of the working conferences are often published as collections of selected and edited papers.

Any national society whose primary activity is in information may apply to become a full member of IFIP, although full membership is restricted to one society per country. Full members are entitled to vote at the annual General Assembly. National societies preferring a less committed involvement may apply for associate or corresponding membership. Associate members enjoy the same benefits as full members, but without the voting rights. Corresponding members are not represented in IFIP bodies. Affiliated membership is open to non-national societies, and individual and honorary membership schemes are also offered.

Design Principles for Interactive Software

Edited by

Christian Gram

Professor of Computer Science,
Technical University of Denmark

and

Gilbert Cockton

Senior Consultant,
MARI Computer Systems

and

Visiting Research Fellow,
Universities of Glasgow and Newcastle-upon-Tyne



SPRINGER-SCIENCE+BUSINESS MEDIA, B.V.

First edition 1996

© 1996 Springer Science+Business Media Dordrecht
Originally published by Chapman & Hall in 1996


ISBN 978-1-4757-4944-1 ISBN 978-0-387-34912-1 (eBook)
DOI 10.1007/978-0-387-34912-1

Apart from any fair dealing for the purposes of research or private study, or criticism or review, as permitted under the UK Copyright Designs and Patents Act, 1988, this publication may not be reproduced, stored or transmitted, in any form or by any means, without the prior permission in writing of the publishers, or in the case of reprographic reproduction only in accordance with the terms of the licences issued by the Copyright Licensing Agency in the UK, or in accordance with the terms of licences issued by the appropriate Reproduction Rights Organization outside the UK. Enquiries concerning reproduction outside the terms stated here should be sent to the publishers at the London address printed on this page.

The publisher makes no representation, express or implied, with regard to the accuracy of the information contained in this book and cannot accept any legal responsibility or liability for any errors or omissions that may be made.

A catalogue record for this book is available from the British Library

Library of Congress Catalog Card Number: 93–83646

 Printed on acid-free text paper, manufactured in accordance with ANSI/NISO Z39.48-1992 and ANSI/NISO Z39.48-1984 (Permanence of Paper).

Contents

Contributors	vii
Foreword	ix
Preface	xi
1 The Context of Interactive Systems Development	1
1.1 Introduction	1
1.2 Terminology	3
1.3 The Development Process	6
1.4 The Development Process: Human Roles	13
1.5 Interactive Software Development Environments	17
1.6 Summary	22
2 External Properties: the User's Perspective	25
2.1 Introduction	25
2.2 Goal and Task Completeness	26
2.3 Interaction Flexibility	27
2.4 Interaction Robustness	37
2.5 Formal Modeling of External Properties	46
2.6 Conclusions	50
3 Internal Properties: The Software Developer's Perspective	53
3.1 Introduction	53
3.2 Internal Properties	55
3.3 Software Techniques	65
3.4 Internal Properties and Software Techniques	74
3.5 External Properties and Software Techniques	78
3.6 Conclusions	88
4 Software Architecture Models	91
4.1 Introduction	91
4.2 A Framework for User Interface Software Architectures	92

4.3	Architecture and External Properties	98
4.4	Architecture and Internal Properties	110
4.5	Conceptual Architectural Models	115
4.6	Example Architectures	123
4.7	Assessing Quality Properties	129
4.8	Conclusion	131
5	Tools and Materials	133
5.1	Introduction	133
5.2	Specification Tools and Materials	138
5.3	Construction Tools and Materials	150
5.4	Commercial Tools	166
5.5	Experiences at Research and Development Sites	179
5.6	Conclusions	185
6	Example: Interface for Air Traffic Controllers	189
6.1	Introduction	189
6.2	The Air Traffic Service	189
6.3	A Simplified ATC Support System	192
6.4	External Properties	193
6.5	Applying the PAC-Amodeus Model	205
7	Conclusions	209
7.1	Predictable Quality?	209
7.2	Contributions	211
7.3	Epilog	214
	Appendix A Glossary	215
	Appendix B Summary Tables	221
	References	233
	Index	243

Contributors

Gregory D. Abowd

College of Computing,
Georgia Institute of Technology,
Atlanta, USA

Gilbert Cockton

MARI Computer Systems Ltd.,
Ashington, and Departments
of Computing Science,
Universities of Glasgow and
Newcastle-upon-Tyne, UK

Leonard J. Bass

Software Engineering Institute
and HCI Institute,
Carnegie Mellon University,
Pittsburgh, USA

Joëlle Coutaz

CLIPS: Communication Langagière
et Interaction Personne-Système,
Fédération IMAG, HCI Group,
Grenoble, France

Michel Beaudouin-Lafon

Laboratoire de Recherche en
Informatique,
Université de Paris-Sud,
Orsay, France

Prasun Dewan

Department of Computer Science,
University of North Carolina,
Chapel Hill, USA

Niels Vejrup Carlsen

Siemens Corporate Research,
Siemens AG,
München, Germany

Alan Dix

HCI Research Centre, School of
Computing and Mathematics,
University of Huddersfield,
Huddersfield, UK

Stéphane Chatty

Centre d'Études de la
Navigation Aérienne,
Toulouse, France

Christian Gram

Department of
Information Technology,
Technical University of Denmark,
Lyngby, Denmark

Keith Hopper

Department of Computer Science,
University of Waikato,
Hamilton, New Zealand

Rick Kazman

Department of Computer Science,
University of Waterloo,
Ontario, Canada

Balachander Krishnamurthy

AT&T Research,
New Jersey, USA

James A. Larson

Intel Corporation,
Hillsboro, Oregon,
USA

Reed Little

Software Engineering Institute,
Carnegie Mellon University,
Pittsburgh, USA

Ian Newman

Department of Computer Studies,
Loughborough University
of Technology,
Loughborough, UK

Laurence Nigay

CLIPS: Communication Langagière
et Interaction Personne-Système,
Fédération IMAG, HCI Group,
Grenoble, France

Sylvia B. Sheppard

NASA Goddard Space
Flight Center,
Greenbelt, Maryland, USA

Helmut G. Stiegler

SNI Siemens Nixdorf
Informationssysteme,
München, Germany

Claus Unger

Department of Computer Science,
University of Hagen
(FernUniversität),
Hagen, Germany

Foreword

IFIP's Working Group 2.7(13.4)* has, since its establishment in 1974, concentrated on the software problems of user interfaces. From its original interest in operating systems interfaces the group has gradually shifted emphasis towards the development of interactive systems. The group has organized a number of international working conferences on interactive software technology, the proceedings of which have contributed to the accumulated knowledge in the field.

The current title of the Working Group is 'User Interface Engineering', with the aim of investigating the nature, concepts, and construction of user interfaces for software systems. The scope of work involved is:

- to increase understanding of the development of interactive systems;
- to provide a framework for reasoning about interactive systems;
- to provide engineering models for their development.

This report addresses all three aspects of the scope, as further described below.

In 1986 the working group published a report (Beech, 1986) with an object-oriented reference model for describing the components of operating systems interfaces. The model was implementation oriented and built on an object concept and the notion of interaction as consisting of commands and responses. Through working with that model the group addressed a number of issues, such as multi-media and multi-modal interfaces, customizable interfaces, and history logging. However, a conclusion was reached that many software design considerations and principles are independent of implementation models, but do depend on the nature of the interaction process.

Therefore, this book concentrates on software principles and properties of interactive systems, and attempts to show developers of interactive systems how to make use of the principles to ensure a high quality user interface.

The report has emerged over some years from discussions at working group meetings, where the members presented their viewpoints on the different topics and supplied drafts of sections and chapters. The drafts have gradually been edited together into this report, which represents a sum of the experience of all the members. As a consequence of this we felt it was not appropriate to develop a full set of references, because it would be very

* The Working Group is denoted 'WG 2.7' in the sequel.

voluminous. Instead, we have tried to select a smaller number of references which we find essential and most relevant for the topics in focus.

Acknowledgements

I would like to thank all working group members for their continuing and untiring co-operation. Nevertheless it was still a hard job for our two editors, Gilbert Cockton and Christian Gram, to cast all the contributions and drafts into a coherent and consistent book. Without their unflagging engagement and enthusiasm this book would never have been published. I also want to thank Carol Larson who patiently and carefully designed and re-designed all the figures, as the manuscript developed.

Hagen, Germany
August 1995

Claus Unger
Chairman, IFIP WG 2.7(13.4)

Preface

Purpose of the Book

The rapid proliferation of interactive systems used by more and more people has resulted in an increasing interest in the *quality* of the user interface of interactive systems. But for any product, quality is an elusive concept difficult to define and difficult to guarantee. As formulated by John Long (1989) ‘The general HCI problem is to design user interactions with computers for desired performance. Performance expresses:

1. The quality of work carried out (product quality).
2. The costs incurred herein, both by the user and the computer (production costs).’

The overall concern must of course be the quality of work, but the very broad formulation of the problem does not try to define what quality means, and it does not address the fundamental role of software in all human-computer interaction (HCI). The software architecture of an interactive system does influence interface quality. This book examines the quality and software engineering problems encountered during construction of reliable interactive systems of high quality. In this context, an *interactive system* is defined as a computer system that interacts with human users.

The ‘Quality’ of an interactive system is broken down into a number of external and internal properties: the former are perceivable, or at least may be inferred, by end-users; the latter are only apparent to the software developer. These properties are then related to the software architecture and to development tools – existing or desired.

The intended audience for this book is primarily software developers constructing interactive systems. Rather than providing universal answers to all questions of development, the book raises wide ranging but coherent issues relevant to the development process. Quality properties are related to software development and it is argued, illustrated by examples, how better quality may be achieved by using the appropriate methodology and tools.

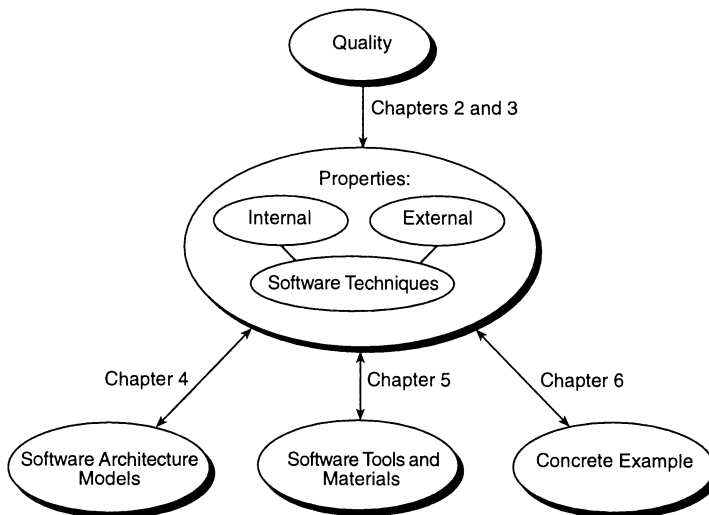
Quality goals of interactive software cover a broad mixture of properties that vary across the many different application domains; some are more related to human factors, others more to software engineering factors. In order to use quality goals constructively they must be measurable. This

is attempted by means of several complementary analyses where we relate quality aspects to software aspects.

This book does not cover the full range of problems, but takes a software engineering approach to the problems of quality and its evaluation. Neither does it cover all aspects of human factors, user models and user evaluation methods; it tries rather to cover a gap between human-factors-oriented results on one side, and software developers' tools and practices on the other side.

Structure of the Book

The book is centered around the set of *properties* considered essential to the quality of an interactive system as illustrated in the figure. Chapters 2 and 3 'unfold' the concept of quality into a number of properties and analyse the interaction between each property and key aspects of software development. The choice of software architecture is seen to interact considerably with the properties, as do software development methods and tools. Chapter 4 links the properties with software architecture in the form of more abstract components, while Chapter 5 takes a 'toolsmith approach' and tries to link the ideas expounded earlier to tools and materials in the real world of the software engineer. To illustrate the approach advocated, Chapter 6 discusses a particular example (an air traffic control system) in some depth, elaborating on the quality properties and showing their inter-dependence in a large and complex system.



Chapters and contents.

Chapter 1 introduces the context of interactive system development: the development process, human roles and development tools. Two principal development models currently in use – Waterfall and V models – are contrasted. The V model stresses the need for a quality plan and quality testing throughout the development phases. Several roles (or actors) are identified and related to the different development phases. The use of development tools is also discussed briefly.

Chapter 2 discusses the concept of quality in interactive systems from the user's point of view, the *external* quality. The highly desirable, but currently unrealizable objective of 'scenario completeness' is analysed and rejected in favor of a more limited set of external properties, which describe different facets of the flexibility and the robustness of the interface.

The properties are not completely independent of each other, but they all contribute to high quality. A formal (state transition machine) model for interactive systems is introduced and is used to suggest some ways of measuring to what extent a system possesses the desired properties.

Chapter 3 looks at quality from a software engineering perspective. Several *internal properties* of good interactive systems are defined. Internal properties are mostly of interest to the software engineer and are not directly visible to, or measurable by, the user. The chapter also identifies a number of software techniques which are of interest during the development of interactive software. A discussion relates the internal properties and software techniques to the external quality of the interface.

Chapter 4 examines the use of software architecture models when constructing high quality interactive systems. The properties introduced in the previous chapters are related to a number of architectures, which are recast into a single framework and are compared with each other. Each architecture emphasizes a certain subset of the properties; this subset is reflected in the structure and purpose of the architecture's components. The components are discussed in the light of the external and internal properties from Chapters 2 and 3, and each property is related to one or more of these components.

Chapter 5 considers the full life cycle of an interactive system and describes engineering tools and materials to be used during the development process. The analysis identifies different forms of interactions between internal and external properties on the one hand, and tools and materials used at stages of the development on the other. The interactions are further illustrated, firstly in evaluations of some commercial development tools and software materials, and secondly in brief style reports that highlight the properties which currently determine the choice of tools and materials at some development and research sites.

Chapter 6 provides a single large example of an air traffic control interactive system to illustrate the properties and the techniques introduced in the earlier chapters. The relevance of each property is discussed in the

light of the specific application, and an architecture for the air traffic control system is proposed.

Finally, Chapter 7 is a short discussion of what is achieved and what is not achieved in the book. It also points out interesting areas for future research related to high quality interactive software development.

The appendices contain a glossary with short definitions of all essential terms used in the book, as well as some of the tables relating properties to techniques, tools and materials.

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

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 sub-phase 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

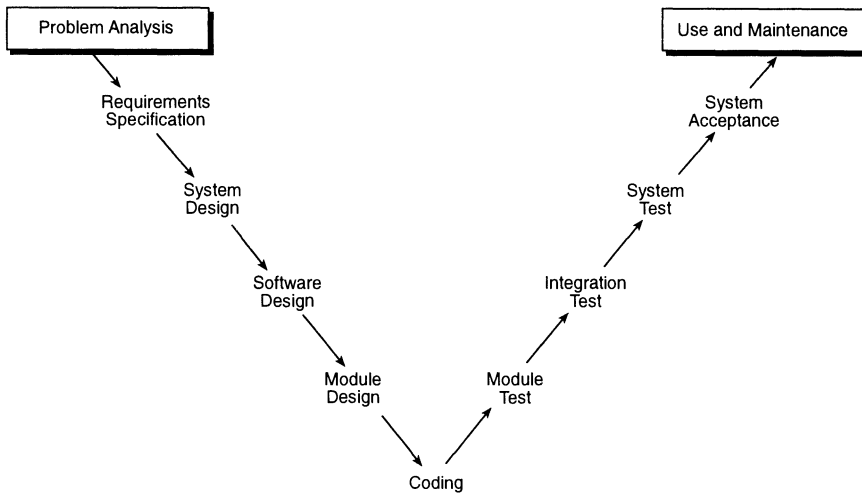


Figure 1.1 *The simple V model for software development. The arrows show the temporal order of the phases.*

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.

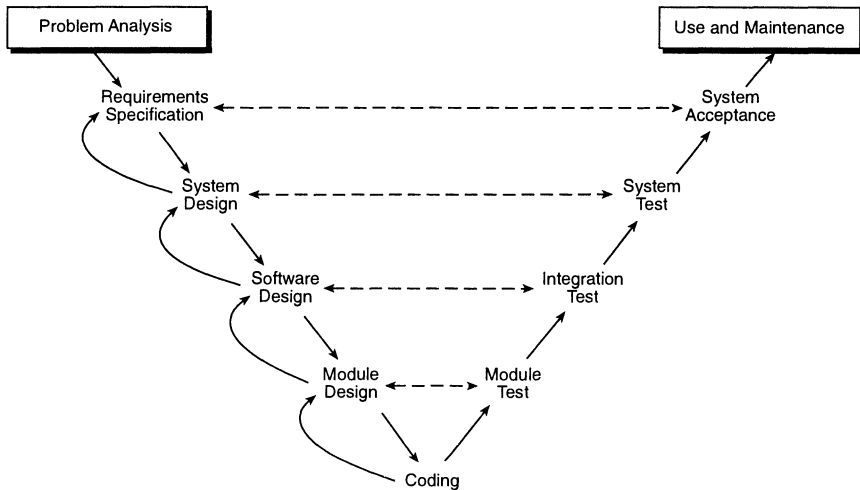


Figure 1.2 *The V model with backtracking. The solid arrows show the usual temporal order of the phases and the backtracking steps. (Backtracking may also jump more than one phase back.) The dashed arrows indicate that test plans must be made for each development phase, and the testing results give feedback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

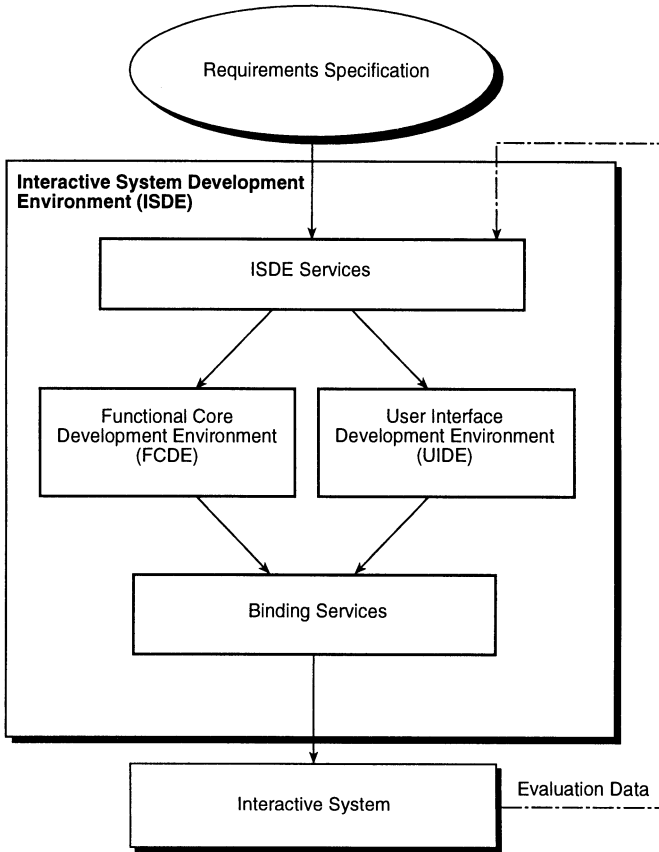


Figure 1.3 *The major components of an ISDE.*

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 play-back; 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

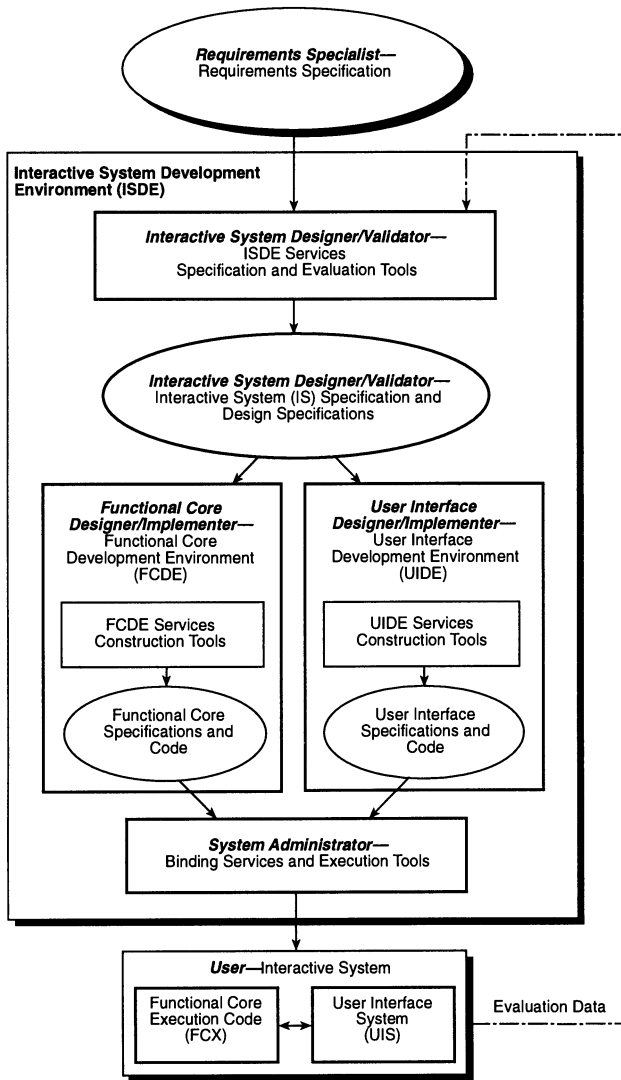


Figure 1.4 The components of an ISDE and their use by the human roles. Each human role is shown in bold face within the components supporting this role.

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.

CHAPTER 2

External Properties: the User's Perspective

2.1 Introduction

The usability of an interactive system is linked to the quality of the dialog, and quality shall here be expressed through a number of measurable properties of the dialog. The aim of this chapter is to identify and define a set of user-centered properties of interactive systems which promote high quality from the perspective of the users. The set must be as complete and mutually independent ('orthogonal') as possible. At the same time these so-called *external properties* must be usable in the software development process as yard-sticks or 'measures' in the quality plan for the development. For a particular system, some of the properties may be absolute requirements (this interactive system *must* have such and such property), while others are desired in a quality plan but are given some 'weight of importance' ($0 \leq w < 1$). Once we understand these properties and their implications, and also the internal properties presented in the next chapter, we will be able to discuss how to construct interactive systems possessing desired and required properties.

Two main approaches are used in this chapter to discuss external properties of interactive systems, informal and formal. The informal discussion, contained in Sections 2.2–2.4, provides a loose characterization of external usability properties as three main principles: task completeness, interaction flexibility and interaction robustness. Flexibility and robustness are further broken down into more basic properties. Each property is defined in natural language and we provide some concrete examples of the property in real systems to help understanding.

Section 2.5 discusses the use of mathematically-based formal models, as an aid towards understanding and distinguishing between various external properties. We do not present a complete formal description of each external property defined in the chapter, and there are no formal proofs of theorems about the properties. Rather, the emphasis is on explaining the kinds of formal models that have influenced our understanding of the external properties and on the distinctions between the various levels of abstraction defined in Chapter 1.

2.2 Goal and Task Completeness

The purpose of an interactive system is to allow users to attain their goals within a specific application domain. If users can reach any goal by means of the system we may talk about *goal completeness*. But this is not a measurable property of the system, it is rather a feature of the combination of users and system, since one cannot foresee every goal a user may form.

For the software developer, it is more relevant to consider *task completeness*, i.e. to ask whether the system supports all adopted tasks (tasks for computer support will be adopted during requirements specification). Successful execution of these tasks when interacting with the system will lead to goal satisfaction. As adopted tasks will only be partially automated, computer support must be shown to meet user needs.

The relevant tasks are found during problem analysis, where future users are studied at their workplace. Descriptions of their work are analysed to isolate common goal states, typical and problematic initial task states, and regular procedures for task execution. This process is an early task analysis, prior to design (but task analyses after design are also important in HCI). It is an essential process in interactive system design and the system should support all of those tasks which have been identified. If we assume an acceptable adoption of identified tasks prior to requirements specification, then task completeness can be defined with respect to the task model. A system is task complete if each task defined in the task model is supported by the system.

Various task analysis methods can be used to generate a task model (Diaper, 1989, and Dix *et al.*, 1993). Tools that are able to generate code directly from a task model, or verify that code actually conforms to such a task model, will increase the likelihood of task completeness.

Note that we do not assume that a user will only ever perform tasks which were predicted by task analysis. Users are far too imaginative and inventive for that assumption to work. As Carroll and Rosson (1991) point out, computer artifacts themselves change the very tasks that users perform. It is therefore impossible to predict all goals or tasks that users may wish that a system supports. This does not, however, reduce the importance of task-driven design for those goals we can predict.

The principle of *task completeness* addresses the question:

- can I do my tasks at all and achieve my goals?

Behind this principle lie the tacit assumptions that users will adapt to the procedures imposed by the system for task execution and that they will make no errors in the process. These oversights can be avoided by asking two further questions about *flexibility* and *robustness*:

- can I do the task my own way? This means that the system should allow user choice during task execution as far as feasible.

- am I supported in doing the task successfully and realizing that I have succeeded? This means that the system should facilitate the user's actions and help the user to recover from mistakes.

These principles are discussed in the next two sections from a task-oriented point of view. The principles do not fully address the complete range of HCI problems. Instead they are a minimal replacement for a universal principle of *scenario completeness*, which addresses the question:

- can all users successfully complete any intended task, regardless of the initial task state (which includes their knowledge and beliefs) and regardless of all expectable events that could arise during task execution?

The principle of scenario completeness has deliberately been left out of consideration because it is not clearly defined and therefore not useful for software engineering. To simplify and even enable subsequent analysis, we have substituted 'adopted goals' for 'any intended task'. The principles of flexibility and robustness address some aspects of typical scenarios, where variations are due to users' preferences, mistakes or slips rather than to the demands of their physical, social and work environments, or to inappropriate beliefs or assumptions.

We have two justifications for this reduced focus:

1. this book reports the first systematic attempt to establish links between what users need and the ways in which software is constructed. It is unreasonable to attempt or expect comprehensiveness in such exploratory work.
2. critiques of task-based design (Bannon and Bødker, 1991, and Benyon, 1992) are still largely polemical, where credible arguments have yet to be backed up by practical consequences. Thus, while we know that learning, users' knowledge, social interaction, working divisions of labour, working practices and situated activity are all relevant to the design of systems, the form of this relevance is not yet clear, and thus we cannot be expected to establish links between what humans need and the ways in which software is constructed.

Our hope is that we have made no commitments to, and depend on no assumptions about theories of human activity that will obstruct future extensions of our framework to incorporate more demanding aspects of human activity.

2.3 Interaction Flexibility

Interaction flexibility refers to the multiplicity of ways in which the user and the system exchange information during task execution. This can apply at all levels of description defined in Chapter 1: functional, dialog, logical and physical levels. Interaction flexibility requires designers to recognize that

people react differently, and designers must respond to user differences and preferences by providing a variety of interaction techniques.

We will list interaction flexibility properties under three subcategories: representation (of information), planning (of task execution) and adaptation (of dialog forms).

Representation of information:

- F1. Device multiplicity** – the capacity of the system to offer multiple input and output devices for communication. Input devices include microphone, keyboard, mouse, dataglove, video camera, etc. Output devices include screen, loudspeaker, force-feedback joystick, etc.
- F2. Representation multiplicity** – the capacity of the system to offer alternative representations for both input and output.
- F3. Input/Output re-use** – the capacity of the system to allow usage of previous input or output as future input.

Planning of task execution:

- F4. Human role multiplicity** – the capacity of the system to support users with different roles.
- F5. Multithreading** – users can engage in several tasks which may overlap in time. In these cases, the system can provide support for the simultaneously active task threads.
- F6. Non-preemptiveness** – Preemption occurs when the system enforces a sequence of interaction that is not necessarily expected by the user. Non-preemptiveness is the absence of preemption.
- F7. Reachability** – the capacity of the system to allow users to reach any system state, regardless of the current state.

Adaptation of dialog forms:

- F8. Reconfigurability** – the capacity of the system to support user-initiated customization of the interaction.
- F9. Adaptivity** – the capacity of the system to initiate customization of the interaction.
- F10. Migratability** – the capacity of the system to support user- or system-initiated transfer of task responsibility.

Reconfigurability and adaptivity are often dealt with as one property, *customizability*, but the software developer must distinguish between functionality that gives the users some choices (reconfigurability) and features that makes the system take the initiative (adaptivity).

The remainder of this section will further examine each of these properties. These properties are again summarized in Table 2.1 at the end of this section.

2.3.1 Device multiplicity

Device multiplicity means that multiple input and output devices can be used for the dialog. For example, in an airline reservation system, the customers might type or speak their queries in natural language. The logical representation in both cases is the same, natural language, but different input devices are used to articulate the query.

Whereas representation multiplicity refers to the dialog and/or logical levels of interaction, device multiplicity refers to the lowest level of interaction flexibility, the physical level; the word device is preferred to other terms used, such as 'media' and 'modality', because they have several interpretations. Some authors refer to a device as a media type. For others, the word media is not constrained to the physical level, but can be understood at higher levels of abstraction (Blattner and Dannenberg, 1992), and the media is understood as a representational system (Alty, 1991). Others, like Bernsen (1993), use the term modality to refer to a representational system. Given this overlap in interpretation between media and modality (which extends to multimedia and multimodality), we will be explicit about our meanings for these terms. Media refers to the device or physical level, whereas modality refers to all other levels of abstraction. Therefore, multimedia corresponds with device multiplicity, while multimodality is a more complex property corresponding with representation multiplicity, the next flexibility property.

2.3.2 Representation multiplicity

Representation multiplicity concerns flexibility in the rendering of state elements as well as in the articulation of input. Multiple representation covers the variation of information content as well as the presentation of the information. For example, on output a system could support alternative representations of the notion of temperature over a period of time. It can be presented as a thermometer, if the actual numerical value is important, or as a graph if it is important to notice trends. It might even be desirable to make both representations simultaneously available to the user. Each representation provides a perspective on the internal state of the system. At a given time, the user or the system is free to consider the representations that are most suitable for the task. When several renderings are presented simultaneously, it is often called multimodality.

Alternatively, a single output on the screen can represent a synthesized representation of a collection of internal system values. For example, performance meters aggregate all system data concerning resource usage and present that information as a timeplot to indicate overall utilization. A quick glance at such a plot can help a user understand performance trends.

An example of input multiplicity would be in a drawing package where

a user may draw lines by direct manipulation or by specifying the end coordinates numerically in text fields. Depending on the task requirements, either means of specifying the input can be important, and both may need to be equally available. If both line and numeric values are simultaneously presented, we have a particularly rich form of interaction referred to as equal opportunity by Thimbleby (1990).

As well as using different representations alternatively they can also be used simultaneously to achieve a certain effect. For example, a sound effect (a 'bip') may be issued as a new window appears.

Representation multiplicity is related to multi-modality but the latter is a more complex issue. Multi-modality also covers how signals on two or more modalities (communication channels) are combined to form a single message.

2.3.3 Input/Output re-use

It is possible to articulate an input expression by referring to previous input or output expressions. Cut, paste and copy commands are typical examples of input and output re-use (I/O re-use). Another simple example of re-use occurs in command line interfaces in which users can select commands from their previous input and re-issue them to the system.

Two interesting issues arise. First, there are implications for type coercion. The second issue is at what level of abstraction is the re-used information interpreted. Type coercion is necessary when the source and target systems use different data structures. For example, the internal representation of a circle differs between object-based and pixel-based graphical editors. Cut and paste re-use will have to support conversion between these different internal representations. Furthermore, it is desirable that such type coercion be invertible, but this is a complicated issue involving interoperability, which we discuss further in Chapter 3.

The issue with interpretation at different levels has implications on how interaction histories are stored by the system. At the physical level, re-use would mean that actual keystrokes would be recorded. Re-use at the functional level would mean that functional-level information (e.g. the command code and the value of arguments) would be recorded. Re-use at the dialog level is rare, as input re-use generally is implemented at the extremes of keystroke input or command invocation. Indeed, it is difficult to identify advantages for dialog level re-use that are not matched and exceeded at the functional level. Aspects of I/O re-use are further discussed in Section 4.3.

Default behavior is related to I/O re-use. Default values are generated by the system based on prior user interaction; they provide input or output based on prior history. Adjusting default behavior based on the interac-

tion history in this way is similar to system-initiated adaptation, called adaptivity below.

2.3.4 Human role multiplicity

In multi-user systems, different users serve different roles or functions in their interactions with the system and, ultimately, other users. A role in this context refers to users' goals and determines the kinds of tasks that they will want to perform, the methods or commands used to accomplish those tasks, and the system objects that will be necessary to complete their actions. A user role is thus identified by a cluster of goals that is allocated to or adopted by a user within some division of labour. Roles may overlap and change over time.

So, for example, in a multi-party conferencing situation, one participant might play the role of the manager whose responsibility is to explicitly pass floor control to the other participants. The participants can assume the role of either the speaker or a member of the audience. Thus, the conferencing system calls for three different roles: manager, speaker and audience. Human role multiplicity refers to the extent to which the system supports the various roles that users can assume.

This variability of roles is most obvious in multi-user systems, but it is certainly not exclusive to groupware. The major distinction between single-user and multi-user systems is that multi-user systems will have to support multiple human roles simultaneously, but also a single-user system must be able to support the user in different roles, sequentially. A simple example of role multiplicity occurs with Hypercard, which has five pre-defined roles that a single user may switch between — scripting, authoring, painting, typing and browsing. Hypercard uses these role definitions both to ease the novice user into more sophisticated programming tasks and to limit some users from changing secure parts of a Hypercard stack. This last concern raises questions about how the system controls access to various system objects, a topic linked to role multiplicity that we discuss under the property of access control for the interaction robustness principle.

Another issue of concern for a system supporting multiple roles is how users are able to change roles. Changes in role can be either user initiated or system initiated, as discussed below under the headings 'Adaptivity' and 'Reconfigurability'. In the conferencing example, it is the duty of the manager to explicitly assign roles to the conference participants. A member of the audience sends a request to the manager to gain control of the floor, but it is up to the manager to make the appropriate role assignments (e.g. reassigning the current speaker to be a member of the audience and assigning the requestor to be the new speaker). So the manager has the capability to reconfigure the system. An adaptive Hypercard system, on

the other hand, might sense that a novice user needs to alter text in a Hypercard stack, and therefore change her role from 'browsing' to 'editing'.

2.3.5 Multithreading

In cooperative systems with several simultaneous users multithreading is a *sine qua non* because the users want to execute their individual task threads in parallel.

In a single-user environment the need for multithreading is less clear cut, and a number of arguments may be brought into play. The rest of this subsection discusses different ways and different levels of multithreading in a system with one user (or a few users).

Often a user wants to do several things in parallel, and if the system has concurrent capabilities, multithreading may help users to achieve their goals. This contributes toward interaction flexibility since it lets users perform multiple tasks simultaneously or switch freely between them. The user may want parallelism on more than one level of abstraction: (i) at the functional level, where parallel *command execution* is possible; (ii) at the dialog level, where parallel *command specification* is possible; (iii) at the logical interaction level, where parallel formation of *elements* of a command specification is possible; and (iv) at the physical level, where parallel formation of *an element* of a command specification is possible.

Multithreading at one level of abstraction implies nothing about threading at a higher level. We can have multithreading at the physical level (e.g. simultaneous input of keyboard and mouse events), but not at the logical interaction level (when they are formed into a single element, i.e. an edited text field value).

Similarly, we can have multithreading at the logical interaction level (e.g. simultaneous formation of options and arguments respectively by keyboard short cuts and mouse selections), but not at the dialog level (where options and arguments part of the same command specification). Lastly, we can have multithreading at the dialog level (e.g. simultaneous formation of command specification in separate windows), but not at the functional level (where command executions are serialized).

At each level of abstraction, the kind of multithreading possible depends on what kind of parallelism the underlying system supports. A system with *concurrent* multithreading allows simultaneous communication of a set of elements that are fully formed at a higher level of abstraction.

A system having *interleaved* multithreading permits a temporal overlap between articulations or specifications but stipulates that at any given instant, only one element of a fully formed structure is being articulated (below dialog level), specified (at dialog level), or executed (at functional level). Concurrent interleaving at one level of abstraction may thus be replaced by interleaved multithreading at a higher level.

Multithreading may also be replaced by single threading at a lower level, but this imposes extra interaction steps on users. Thus in a typical window system, the interaction is interleaved multithreaded at the dialog level, as the user may communicate interleaved with a number of open windows supporting different tasks, but at the physical level the user interacts with one window at a time in a serial manner, utilizing one mouse and one keyboard only. This imposes explicit changes of focus on users. Similarly, operating system command languages with backgrounding or job control are multithreading at the functional level, but command specification at the dialog level is single-threaded, forcing the sequential articulation of operations, options and parameters. This imposes explicit articulations of backgrounding and foregrounding on the users.

Just as serialization below interleaved multithreading complicates interaction traces (by adding explicit changes of focus), so concurrent multithreading is more simple than interleaved multithreading, as it avoids suspension and resumption of interrupted formations, and avoids explicit backgrounding and foregrounding.

As an example at the physical level, consider two acts for a user of a computer painting application: changing brush width and painting on the canvas. In a single-threaded or interleaved multithreaded-system, users would have to suspend painting acts in order to change brush width. In a concurrent multithreaded system, such as VoicePaint (Gourdol *et al.*, 1992), users could change the size of the brush as they are painting, and thus be able to paint a crescent moon in one uninterrupted stroke of a locator device.

Designers must carefully consider differences in 'threading' between levels of abstraction, since transitions from single to multithreading and from concurrent to interleaved multithreading require users to plan their interactions more carefully. Interaction is both more flexible and more simple when multithreading is concurrent at all levels of abstraction in interactive systems.

2.3.6 Non-preemptiveness

When considering the interaction between user and system as a dialog between partners, it is important to consider which partner has the initiative in the conversation. The system can initiate all dialog, in which case the user simply responds to requests for information or action. We call this type of dialog system-driven because the system more or less decides which action (or actions) the user may perform next. Alternatively, the system might only react to user input, in which case the dialog is called user-driven because the user has more freedom in choosing the next action. A dialog where either the user or the system may have the initiative is called a mixed-initiative dialog.

Non-preemptiveness refers to the degree of freedom the user has in deciding what next action to perform at the interface, and it is one of the key factors contributing to the user's feeling of flexibility in the dialog. System-driven models of interaction tend to be preemptive, they limit the user's choice of next available action, whereas user-driven interaction favors non-preemption.

Preemptive systems limit the user's options for communication. For example, a dialog box may prevent the user from interacting with the system in any way that does not direct input to the box. From the user's perspective, a system-driven interaction hinders flexibility whereas a user-driven interaction favors it. In general, we want to minimize the system's ability to preempt the user although some situations may require it for safety reasons. In situations in which a user error or slip would result in serious damage without a chance for recovery, it is desirable – or even necessary – to limit user freedom.

The task analyst must have a good understanding of the sets of tasks the user is likely to perform with a system and how those tasks are related in order to minimize the likelihood that the users will be prevented from initiating or advancing some task at a time when they would want.

2.3.7 Reachability

Reachability refers to the possibility of navigation through the system states. It can be defined at any level of detail, but in this context only observable states are of interest. Various aspects of reachability have been given formal definitions, but the main notion is whether the user can navigate from any given observable state to any other observable state. From the user's point of view it may be useful to distinguish between backward and forward reachability.

The user may want *backward reachability* in order to get back to some previous state of the interaction, after having made a mistake or realizing a need for some previous information. This type of reachability is covered by the property of recoverability, as it is usually defined, and it requires sufficient history information to be kept by the system.

Forward reachability means that the user is able to proceed to any desired interaction state, independently of previous dialog development.

These coarse definitions say nothing about how difficult it is to go from one state to another. For example, in order to make a text editor fully reachable, we only need to provide the ability to insert letters sequentially and erase the entire contents of the editing buffer. Such an editor will be reachable because any possible buffer contents can be achieved from any other buffer state. However, the only way to effectively delete the last character would be to erase the whole buffer and type in the entire contents again! To get around this over-simplified reachability criterion, we

can invoke Thimbleby's principle (1990) of *commensurate effort* — things that are easy to do should be easy to undo. Since it is easy to mistype a character in a text editor, it should be just as easy to undo that error (e.g. by providing a delete character command).

Therefore, a more general definition of reachability should include some measure of the ease of navigating. A system has good reachability if the user can navigate from one observable state to another with an effort which in some sense is commensurate with the user's expectation.

2.3.8 Reconfigurability

Reconfigurability refers to the user's ability to adjust the form of input and output. This customization may be very limited, with the user only allowed to adjust the position of soft buttons on the screen or redefine command names. The power given to the user can be increased by allowing the definition of macros to speed up the articulation of certain common tasks. In the extreme, the interface can provide the user with programming language capabilities, such as the Unix shell or the script language Hypertalk in Hypercard.

Such user-initiated changes can also have varying periods of duration. For example, changes could be limited to one interaction session or they could be recorded and affect all future sessions (e.g. resource settings in the X Window system).

2.3.9 Adaptivity

Adaptivity is automatic customization of the user interface by the system. Decisions for adaptation can be based on user expertise or observed repetition of certain task sequences. The distinction between adaptivity and reconfigurability is that in a reconfigurable interface the user plays an explicit role in customization, whereas his role in an adaptive interface is more implicit. A system can be trained to recognize the behavior of an expert or novice and accordingly adjust its dialog control or help system automatically to match the needs of the current user. This is in contrast with a system which would require the user to explicitly classify themselves as novice or expert at the beginning of a session (Kuehme *et al.*, 1992).

Automatic macro construction, as proposed in the Eager system (Cypher, 1991), combines reconfigurability with adaptivity in a simple and useful way. Repetitive tasks can be detected by observing user behavior, and macros can be automatically constructed from this observation to perform repetitive tasks automatically.

2.3.10 Migratability

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

Table 2.1 *Summary of interaction flexibility properties. The 'Description' column contains a short description of each property; the 'Related properties' column is a reminder of relationships to other properties mentioned under each property*

Flexibility Property	Description	Related properties
Representation:		
Device multiplicity	More than one way to do something	Multi-media capability
Representation multiplicity	More than one way to present something	I/O multiplicity, equal opportunity, multi-modality
Input/Output re-use	History repeating itself	Use of defaults
Planning:		
Human role multiplicity	Several people doing several things	Access control
Multithreading	One person doing several things	Concurrency, interleaving
Non-preemptiveness	Doing what you want when you want	User-driven dialog, mixed-initiative dialog
Reachability	Getting anywhere from anywhere else	Commensurate effort
Adaptivity:		
Reconfigurability	The user changing the interaction	Programmability of the interface
Adaptivity	The system changing the interaction	Automatic macro construction
Migratability	Transferring control	

As for many other properties, migration can occur at multiple levels of abstraction. At the physical level, the provision of command completion

migrates responsibility for some physical operations (typing) from the user to the system. At the dialog level a system with number input could allow the user to enter not only literals (e.g. 24) but also numerical expressions (e.g. 6×4). Here a step that would be a user calculation in a full cognitive task description is migrated to the system. At the functional level, file saving is an example of a migratable task: one user may do it explicitly, and another user may want automatic file saving. Interestingly, the effect of migration at the functional level is to reduce the number of commands and objects that users are effectively in contact with during subsequent interactions.

2.4 Interaction Robustness

An interactive system is called robust if it supports a user in performing a chosen task without irreversible mistakes, and if it gives users a correct and complete picture of task progress. Thus, interaction robustness covers all those properties that minimize the risk of task failure. We identify seven properties that contribute to the principle of interaction robustness. The first three properties ensure a correct and complete picture of the system, while the last four properties lessen the risk and cost of mistakes.

Correct picture:

- R1. Observability** – the system makes all relevant information potentially available to the user.
- R2. Insistence** – the dialog structure ensures that necessary information is perceived.
- R3. Honesty** – the dialog structure ensures that users correctly interpret perceived information.

Few mistakes (no irreversible ones):

- R4. Predictability** – users can predict future states and system response time from the current and prior observable states.
- R5. Access control** – the system allows for defining control policy and availability for information access.
- R6. Pace tolerance** – the system allows users to control the pace of interaction.
- R7. Deviation tolerance** – the system supports users' correction of slips and errors.

The first four of these properties – Observability, Insistence, Honesty and Predictability – are very user-dependent; they can only be validated by user testing. The last three – Access control, Pace tolerance and Deviation tolerance – are less user-dependent and can be validated reasonably within

a system (either by analysis of specifications or by expert walkthrough of an implementation).

The remainder of this section will further examine each of these properties. These properties are again summarized in Table 2.2 at the end of the section.

2.4.1 Observability

A system is observable if it allows users to inspect all information relevant to their tasks. This does not necessarily mean that all relevant data are presented at once. A typical screen-based computer system can only render a small amount of the total information on an output device. Hence there must be some browsing function allowing the user to inspect the information in stages.

An important part of observability is to restrict it to all *relevant* information. It could be argued that one would like to be able to view all of the system state, but even if this were possible the user might suffer information overload. So, in large industrial control rooms and aircraft cockpits where vast numbers of dials have traditionally displayed all of the state, there is an emphasis on glass displays which focus the operator's attention on parts of the available information.

We want the immediately perceivable information to be relevant and sufficient for the user's current tasks. The task model should contribute to this design issue, by identifying those elements of the system state which are most critical at each interaction point. Having identified these elements, lower levels of design can ensure that the identified elements are indeed rendered at the required times. In a constrained process this task-based identification may be sufficient, but in most systems the user will have some control as to which elements are displayed, for example in choosing the positioning and visibility of windows.

Clearly in most situations only the most critical information can be immediately available. However, the user should be able to access all relevant information eventually. This kind of observability – often called *browsability* – is based on the general principle of allowing the users to perceive anything they can name, i.e. anything he or she can provide a description for. If the description is not ambiguous, then the required information should be provided; if it is ambiguous, the user should be given sufficient information about possible choices in the present state. This means that the system must present the information and the possible actions which seem to be useful. Furthermore, this browsing of information should be possible without modifying the system state (other than the form and contents of the current presentation). This requires some kind of multithreading allowing the user effortless return to the state from which the browsing was started.

It is particularly important that a user is aware of the effect of the last

action. That is, that there is effective feedback. Ideally, all changes should be immediately perceivable, but where this is not possible (for example, in a document-wide search/replace), at least some indication of the effects should be given (for example, displaying the number of changes that resulted from the search/replace). In addition, a user ought to be able to browse the effects of the previous action, but this is frequently inadequately supported. For example, after a search/replace the user ought to be able to find out precisely which words were affected, but this is usually impossible.

In an open system like a CSCW system,* the user will also want to be aware of the actions of other users and of external processes. This observability of other users' actions has been referred to as *feedthrough* (Dix, 1994). In practice, we may accept weaker feedthrough mechanisms than those for feedback. For example, in a shared editor feedback of a user's own typing must be virtually instantaneous, whereas network delays of a few seconds may be acceptable for feedthrough. However, as the user did not initiate the actions, merely seeing the effect may be insufficient; it is frequently helpful for feedthrough to identify both the action which caused the change and the user who initiated it. This can be very important in allowing users to interpret the intention behind other users' actions.

The observability of a system is – like the predictability discussed below – tied to the user's understanding of the system and their expectations of its behavior.

2.4.2 Insistence

Just because information is available at the interface, it is not necessarily the case that the user will notice it. For example, one persistent problem in windowed systems is the situation where the user mistakes which window is selected and then directs text at the wrong window. All window systems give some clue as to the active window, often by highlighting the window's name, or emphasizing its border. However, if the user is looking at the content of the window, these borders may be insufficiently salient.

Software designers should do their best to ensure that critical information is not only available, but is actually *perceived* by the user. In addition, the system must ensure that events are reported and noticed at the appropriate time. In particular, if an event indicator is ephemeral (e.g. a buzzer) and the user is temporarily absent, then the user may never realize that the event has occurred.

Insistence can be achieved by various means: by increasing the visual salience, by interrupting the user with pre-emptive dialogs, by using aural signals or by leaving persistent event indicators. The choice of mechanism

* CSCW = Computer Supported Cooperative Work.

is again finely dependent on the plausible cognitive task description, which should identify two aspects:

- where the user's attention is likely to be (in order to assess the salience of a particular feature);
- the required timeliness and salience of different system elements.

Different interface widgets have different salience and timeliness properties. For example, a buzzer may demand instant attention whereas the appearance of a new icon may be eventually noticed after a few minutes. Too many over-salient features will lead to a noisy and unpleasant interface and furthermore will hide the features which are really important. The designer should therefore attempt to match the properties of the interface widgets to the required salience and timeliness required by the user's task.

In an open or multi-user system it is also important that the user is made aware of appropriate events generated by other users and the environment. For critical events this should be very salient, but it is also useful to generate a low-level, ambient indication of external activity. This is called *awareness* in CSCW systems. Experiments have shown that this can be very important in giving users a sense of working together and in diagnosing changes in the world (Gaver and Smith, 1990).

2.4.3 Honesty

Designers need to ensure that users interpret the symbols at the interface in the way that they are intended. In Norman's terms (1988), this is called the 'gulf of evaluation'. Maybe a system is observable and insistent, but if users misinterpret the information, there is bound to be trouble. Honest systems strive to achieve a match between the user's interpretation and the designer's intended interpretation of the interface. This requires that (i) the observable state conforms with and represents the relevant features of the system state, and (ii) the user interprets the rendered information correctly. The second part of this can only be validated through user testing, but the first part is the responsibility of the designer.

Various heuristics exist for promoting the honesty of a system. The notion of affordance from psychology deals with how artifacts in the real world (including a computer's interface) suggest the correct mode of operation. For example, buttons should suggest pushing them to get some operation. Mouse cursors should suggest pointing or dragging. Metaphors are frequently used in an interface to borrow from a user's previous knowledge of how things behave in the world. If a designer structures the interaction consistently according to some clear metaphor to the user, the user's familiarity with the metaphor will assist them in guessing the correct behavior. The potential danger with metaphors in a computer interface is that they might suggest operations which are not possible, or they might suggest the

wrong interpretation of an action (e.g. dragging the icon for a floppy disk, or the hard disk, to the trash icon in the Macintosh desktop metaphor).

Observability and insistence are robustness properties that support honesty, but also flexibility properties may improve the honesty of a system. As an example, if a system is reconfigurable (the user may change the interaction), users can make changes that make the system more honest to them; and an adaptive system (the system changing the interaction) may adapt to a user's habits and thereby be perceived as more honest.

2.4.4 Predictability

The above discussions of observability and insistence focus on the extent to which the system provides enough information for the user to know how past actions have affected the present state. Honesty refers to how this information is correctly comprehended by the user. Predictability concerns the future, that is, to what extent information in the past and present can help the user determine the outcome of future interactions. Like the first three robustness properties predictability depends not only on the system itself but also on the actual user knowledge and expectations. But the designer can do much to further (or impede) the predictability as perceived by the users.

Roughly speaking, predictability means that a user's knowledge of the past interactions and current observable state are sufficient to determine future behavior of the system. The system must be designed to provide state and action information in a reasonably complete, systematic and consistent way. Finer measures of predictability are concerned with just how much of the past is necessary and to what extent behavior is predictable based on immediately observable information.

Consistency is one heuristic that is often applied to increase the predictability of an interface. Consistency allows the user to generalize from specific situations to similar situations. But it is difficult at times to determine, at design time, which situations a user will consider similar or dissimilar.

Predictability is about actions as well as effects. Users need to know not only what will happen when they issue a command, they need to know what commands are available to them at any point. Several properties we have discussed already are related to this property of operation visibility. Affordance means that the operations which are suggested to a user are the ones that are available, that is, affordance means honesty with respect to operation visibility. If the user sees a button that says cancel, is it really the operation that is available? And what will be cancelled? This may depend on the user's actual role, as discussed in Section 2.4.5; when users play limited roles they should know that their set of possible actions is limited.

Users also expect the system to be stable with respect to response times.

A predictable system must therefore exhibit *temporal stability*: if the same action – for example, opening a new file – is executed several times, the response time should be the same each time. Furthermore, execution times for similar tasks (i.e. tasks which the user perceives as similar) must be close to each other.

2.4.5 Access control

Access control mechanisms restrict those parts of a system which a user can view or alter. This is particularly important in systems where the users can assume multiple roles. In a groupware application supporting synchronous editing, certain users could be designated as the editors of parts of the text. Users who are not editors of part of a text should not be allowed to make changes to that text, though they may be allowed to view the text.

Access control addresses the robustness concerns brought about by human role multiplicity. In a single-user context, the actions of users in some roles might be restricted in order to prevent damage to the system. Users in other roles might have very few restrictions placed on their interaction. Though there are good reasons to allow such free interaction, it is not without risk. For example, a Unix super-user has free reign, which means that most operating system safeguards which apply to normal users (e.g. file ownership) are circumvented. Whereas the power of the super-user is necessary for some tasks, it means that the system is less forgiving of slips at the interface.

A simple example of an access control mechanism arises in most modern operating systems, in which protection schemes are used to tailor access to files and control who can alter file contents. In a multi-user text editor, users in the role of commentator are prevented from changing the text owned by some author, though their comments are made visible to editors and authors.

Access control is not always motivated by the desire to prevent users from doing bad things. The training wheels metaphor (Carroll and Carrithers, 1984) in which a system adjusts the availability of functionality based on a user's level of expertise, is intended to ease the learning burden for the novice. Rather than risk exposing a new user to the daunting variety of potential functions, the system begins by offering a minimal set of functions, gradually revealing more functions as user experience grows. This gradual revelation of system functionality does limit the user's access to the system but with the intention of decreasing anxiety and increasing learnability.

In multi-user systems, it is important that users be aware of other users' activity, even if they cannot affect their actions directly. We saw an example of this earlier in the multi-user editor with the visibility of comments. This is related to feedthrough (see Section 2.4.1). In a shared graphical editor, each user may be limited to actions on their own private painting layer.

The system superimposes all layers to reveal to each user what the overall shared picture looks like. Though individual users cannot alter any drawing on another user's layer, they can still see what the others are doing and can coordinate their drawing activity accordingly.

When a user's actions are limited by some access control mechanism, it is also important that they understand the scope of the access control. Again, in the multi-user editor a single author not only wants to know what parts of a document are available to them for editing but also wants to know what other parts of the document are open to others for editing, indicating which parts of the text are liable to change. For example, consider the design of media spaces (i.e. computer mediated video for computer-supported co-operative work), where users can generally make arbitrary live video connections with other users. Privacy issues demand that a user be able to prevent arbitrary intrusion. When someone wants to make a connection with another user and that connection is denied, the system could indicate whether the person is busy with another video connection or is restricting access to their office for the time being.

2.4.6 Pace tolerance

Interaction take place in the real world. There the time it takes for things to occur matters. It takes time for both the user and the system to react to changes in the world. A pace-tolerant system considers the timing match between user expectation and system demands. For example, it takes time for a user to read a message, so sending numerous error messages to screens which scroll away faster than they can be read is a timing mismatch. Designers of such systems fail to realize that what is a quick task for the system is not so quick a task for the user.

As technology improves, we may be led to believe that the system will always be able to keep up with the user. Dix (1991) points out that adherence to this myth of the infinitely fast machine would lead us to conclude that we only ever have to worry about slowing the system down to the user's level. No matter how our technology progresses, it will always be the case that things take time. External factors (network latency, hardware failures, etc.) and increasing system demands (load the entire Oxford English Dictionary when doing a spell check!) will always mean that we should design the system knowing that delays will be present. Pace-tolerant design is conscious of how to meet the user's expectations both when the system is too fast and too slow. A standard concession to the 'faster' user is to provide type-ahead; while the system is busy doing some other task, the user is allowed to provide input to the system that will be interpreted once the system is available. Type-ahead acknowledges that there are situations in which the user works faster than the system.

How to interpret typed- (or clicked-)ahead commands is a tricky issue,

especially in a graphical user interface and it points out the criticality of pace-tolerant design. It is very easy for a user to click at several soft buttons quicker than the system can respond to them. Suppose a user sends a command to launch an application in a new window. While waiting for the new application to initialize and appear on the screen, the user decides to send some commands to other windows. The system stores the user actions to be interpreted once it has finished the task of launching the new application. In what context are those actions interpreted once the system is ready to respond to them? The reasonable answer is that they should be interpreted in the context in which they were issued, but if all that is stored in the type-ahead buffer are physical actions (keystrokes, mouse clicks and mouse positions), then it is virtually impossible to guarantee that the actions will be interpreted in the correct context. Pace-tolerant type-ahead is not a solution that can be implemented at the physical level alone, but involves higher levels of abstraction as well.

When interface behavior is time-dependent and the user is not aware of the dependency, the interface will be hard to learn. How difficult is it for users to understand the difference between a double-click of the mouse and two separate mouse clicks? And once they understand the difference, will they expect a different system interpretation for two mouse clicks than for a single one? If users don't expect the speed of their interactions to affect the interpretation, then they probably won't learn on an interface that relies on it.

2.4.7 Deviation tolerance

No matter how well a system has been designed, users will commit errors from which they will want to recover. Deviation-tolerant systems may support (i) detection of error states or 'dangerous' states, (ii) prevention from getting into error states and (iii) correction of slips and errors. When users can rely on being warned against dangerous actions and being assisted in recovering from small errors, then they will feel more free to explore an interface without worry, i.e. the user will experiment with the interface with the expectation of being able to undo some operation once he or she has learned how it works.

Even in a system with a very careful design of mechanisms for detecting error states and for prevention, the users will commit errors. Therefore the most important aspect of deviation tolerance is the provision of recovery procedures for the user. Recovery is a special form of reachability; the user wants to get from some state of interaction to another. The initial state for recovery is an error state (an unwanted state) and the final state is the corrected state. There can be two different strategies for recovery – backward and forward. A backward recovery strategy, such as undo, has a previously attained state as the corrected state. A forward recovery strategy, such as

Table 2.2 *Summary of interaction robustness properties. In the second column, a '+' indicates the property is tied to user expectation and that validation depends on user testing. The 'Description' column contains a short description of each property. The 'Related properties' column is a reminder of relationships to other properties mentioned under each property*

Robustness Property	User dep.	Description	Related properties
Observability	+	The user may perceive	Immediacy, browsability, feedback, feedthrough
Insistence	+	The user will perceive	Salience, timeliness, persistence, awareness
Honesty	++	The user correctly comprehends	Affordance, familiarity, suggestiveness, guessability
Predictability	+	Understanding how the system will react	Observability, consistency, affordance, response time stability
Access control		Role-sensitive restriction of information availability	Human role multiplicity, feedthrough, awareness, visibility, privacy
Pace tolerance		Response times match user's expectations	Timeliness, adaptivity, migratability
Deviation tolerance		User's recovery intentions are supported	Forward/backward recoverability, commensurate effort, pre-emptiveness

negotiation in a cooperative system, selects a previously unattained state as the corrected state. In some cases only one of the strategies will be available. When both strategies are available, it is the user who decides which to adopt. Recovery should be viewed, therefore, as a user intention, not as a function provided by the system. The designer must make the system deviation tolerant by supplying understandable and easy 'escape routes' from anticipated unwanted states, not by building one recovery function.

Thimbleby's notion (1990) of commensurate effort is again important here. In this situation it means that if an error is easy to commit, its effect must be easy to recover from.

The robustness property of deviation tolerance must be balanced against the flexibility property of non-preemptiveness. A non-preemptive system 'allows the user to do anything in any case', while a deviation-tolerant system 'guides the user away from dangerous slips and errors' and in case of a

slip ‘guides the user towards safe recovery’. For example, a non-preemptive power station control system would allow the operator to close a cooling water pump without further ado, but a deviation-tolerant system would react with a modal warning saying ‘you can’t do this unless you also...’.

2.5 Formal Modeling of External Properties

The external properties cover so wide a spectrum that no single formal model can be used to define and discuss all of them. Some of the properties are tied intimately to the users’ perception and behavior, henceforth they are difficult or impossible to formalize.

In this section, we present some simple formal modeling activities which lead to a clearer understanding of some of the external properties associated with usability. But our formal model does not contain real time, and therefore the temporal aspects of a dialog are not formalized below.

In Chapter 1 we introduced four different levels of abstraction for an interactive system: the functional, the dialog, the logical and the physical level. At each of these levels we describe the interactive system as a deterministic state machine, or labeled transition system, consisting of a set of states (system states), a set of events and a function which relates events to state transitions. A state machine, \mathcal{M} , is a 3-tuple

$$\mathcal{M} = (S, E, \rightsquigarrow),$$

where

- S = the set of possible states the machine can assume;
- E = the set of events or operations the machine can engage in, sometimes referred to as the alphabet;
- \rightsquigarrow = the transition function, which maps events in E to transitions on S . The signature is: $\rightsquigarrow : S \times E \mapsto S$.

We are thus introducing four state machines, one for each level of abstraction, and below we shall further introduce the set Obs of observable states, i.e. the states which the user can perceive and distinguish.

But first we use the general machine model \mathcal{M} to illustrate the difference between some of the properties described informally in Sections 2.2–2.4. For example, we can distinguish between reachability and non-preemptiveness. To make this distinction, we will appeal to a graphic depiction of a state machine, or state transition diagram, like the example shown in Figure 2.1.

2.5.1 Flexibility properties formalized

Reachability refers to the connectedness of the state transition graph. Initial reachability means that the user can get from the initial state to any other state in the system. In the example in Figure 2.1, with $S1$ as initial state, the

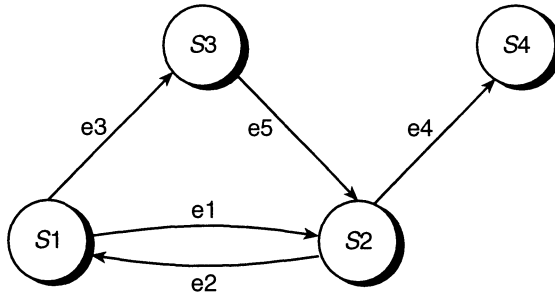


Figure 2.1 A transition diagram for a simple dialog with four observable states. $S1$ is the initial state, and $e1$ – $e5$ are events triggered by user actions.

dialog has this initial reachability property. In the more general formulation of reachability we ask whether it is possible to get from any state to any other state. In the example, this form of reachability is not satisfied, since there is no event transition from state $S4$ to any other state in the system.

Non-preemptiveness, on the other hand, is not just asking about connectedness, but about the shortest paths between states. It can be defined more or less fine-grained. Complete non-preemptiveness means that the transition diagram is fully connected, i.e. there is a direct path from any state to any other state. Such a requirement does not make sense in a real system of any complexity, and therefore a system design must include decisions on the degree of non-preemptiveness required.

Whereas there is a path from state $S3$ to $S1$ in Figure 2.1, it is only possible to get to $S1$ by first going through $S2$. This intermediary state is pre-emptive. This example demonstrates that non-preemptiveness is relative; it depends on what actions or tasks the user wants to perform. If the user never would want to get to $S1$ from $S3$, then this pre-emption is acceptable (and maybe even helpful by preventing unwanted actions).

Multithreadedness refers to the possibility of having independent threads of activity going on at the same time, and it can only be expressed here by having several state machines acting in parallel. A formalism capturing this must involve synchronization mechanisms or real time in some form (as for instance CSP* or Petri net). But at the functional level, we can identify the tasks that the user can be performing, and from this functional perspective a multithreaded system will allow interleaving of those functional tasks.

Input/output re-use can be described in the state transition model at the dialog or logical level. Re-use is possible if there is a transition from a state rendering some input/output value to a state using that same value as input for another task.

* CSP = Communicating Sequential Processes, a language for parallel programming.

2.5.2 Robustness properties formalized

Deviation tolerance, or recoverability, is in state model closely related to reachability (discussed in the previous subsection) and the connectedness of the state graph. Deviation tolerance is examined by determining which states in S are error states and whether there are ‘undo’ paths back to ‘normal’ states.

Observability can be discussed using the different levels of state machines and the concept of interaction points. The machines form a hierarchy, where the functional level machine \mathcal{M}_F is considered an abstraction of the dialog-level machine \mathcal{M}_D , which is an abstraction of the logical-level machine \mathcal{M}_L , etc.

So we have abstraction functions between the states of each machine:

$$\text{Abstr}_P^L : S_P \rightarrow S_L \quad \text{Abstr}_L^D : S_L \rightarrow S_D \quad \text{Abstr}_D^F : S_D \rightarrow S_F$$

and sequences of events (transitions) at one level can be interpreted as a single event (transition) at the level above. Figure 2.2 illustrates two of these levels of abstraction.

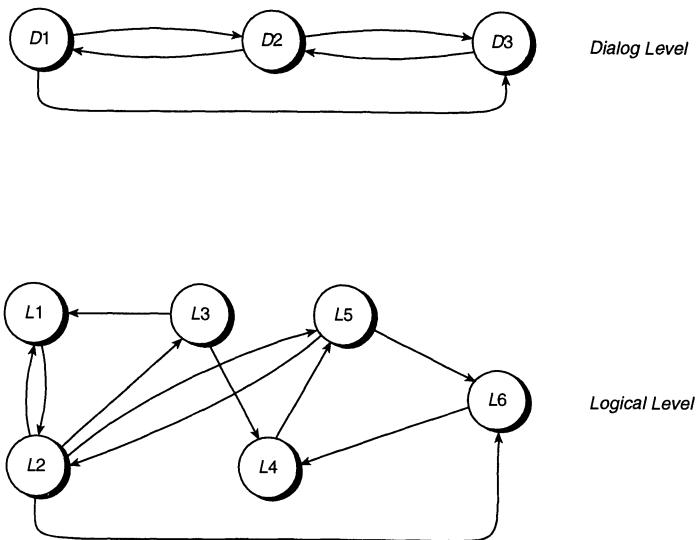


Figure 2.2 Two levels of state machines. The dialog-level machine has three states, $D1$, $D2$ and $D3$, and the corresponding refined logical-level machine has six states, where $D1$ is the abstraction of $(L1, L2)$, and $D2$ is the abstraction of $(L3, L4, L5)$.

A specific dialog can be described by its state trajectory, i.e. the sequence of states activated during the dialog. An *interaction point* is defined as the point at which a sequence of events at one level can be interpreted as an

event at the next higher level. (At the highest level, the functional level, all states represent interaction points.) Events between interaction points at one level do not cause state transitions (are not ‘seen’) at the next higher level. Figure 2.3 depicts this relationship between states and events at different levels of abstraction.

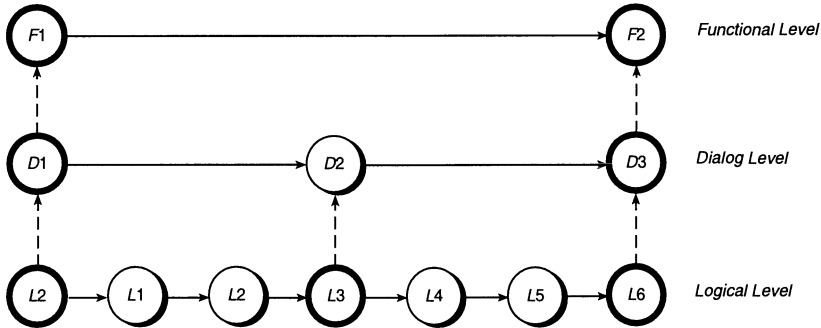


Figure 2.3 *Three levels of state machine trajectories. The interaction points are marked with thick circles.*

The user observes and interacts directly with the physical-level machine, at the ‘keystroke level’. But usually only a part of the internal system state is rendered to the user. What the user can perceive is some projection, or rendering, of the physical system states. This may be described by a rendering function from the set of physical states to the set of observable states:

$$\text{render} : S_P \rightarrow \text{Obs}$$

We can now formulate *observability* properties in this multi-level model. The user only perceives information rendered at the physical level. This level contains information about the higher levels (by means of abstraction/refinement). Therefore, we can ask to what extent the observable information covers all of the dialog or functional information. If the rendering doesn’t reveal all information of higher levels, we can ask whether it is possible to browse at the physical or logical level, a strategy in which physical events (such as scrolling a window) provide different observations of the logical state without changing the corresponding dialog or functional states. That is, a passive browsing strategy is one in which the logical level activity occurs between interaction points of the higher levels.

Even if the property of *predictability* is very dependent on the user’s perception, we may be able to say something about it in terms of the multi-level state machine model. It is a deterministic model where, given a state $s \in S$, any event $e \in E$ leads to a unique new state $s' \in S$. (This corresponds to modeling \leadsto as a function.) But the user does not always

know or understand how the system will behave. The uncertainty – or nondeterminism – arises because the system cannot tell the user everything about its state. The *render* function above defines the observable state space *Obs*, and predictability must be formulated with respect to what the user can observe.

A necessary condition for the system to be predictable from the user's perspective is that there is a function which maps events to deterministic transitions on the observable state. It is important to note here that this formulation of predictability is a necessary condition, but not a sufficient one. That the observable effects of events are deterministic does not guarantee that the users will actually perceive the determinism. The formalism can only suggest that the definition of the observable state space *Obs* be such that we can guarantee that users perceive it. Here the property of *insistence* and the persistence of information in *Obs* will play a role and influence how predictable the system occurs to the users.

2.5.3 Summary of formal model of external properties

This section has demonstrated how some of the external properties presented in this section can be better understood by attempts to model them mathematically. We do not claim to be able to provide such models for all of the properties given in this chapter, nor would such an exercise necessarily be beneficial. The formal models are only useful to the extent that they make clear distinctions between properties or suggest new properties, or if they can be used in verification of a system.

2.6 Conclusions

In this chapter, we have presented a catalog of external properties of interactive systems which characterize usability, and which can be useful in the software development process as yardsticks for quality. To summarize, the external properties fall into three categories:

Goal and Task completeness – you can do what you thought of doing.

Flexibility – you can do things in several ways.

Robustness – you can avoid doing things you wish you hadn't done.

The last two principles attempt to compensate for obvious limitations of Goal and Task Completeness, by considering user preferences and planning, their need to understand the state of a system, and likely sources of error and frustration. These extensions still fail to cover the full spectrum of requirements of the more demanding principle of 'scenario completeness' (page 27). However, there are also principles that could be adequately addressed without covering all the requirements of scenario completeness. Some notable examples of omissions are:

Learnability – the ease with which novice users achieve competent performance with new systems.

User satisfaction – how a system makes the user feel in terms of sense of accomplishment or excitement.

Rather than suggest that we could present a complete catalog of external properties to support usability, we consider the properties discussed to represent (some aspects of learning and user satisfaction apart) the current state-of-the-art. The properties are defined to form as complete and ‘orthogonal’ a space as possible, but a number of interdependencies and trade-offs will show up in Chapter 6 when discussing concrete examples. We have introduced a finite-state machine model to illustrate how at least some of the properties may be formalized, but formalization has not been carried through, because it is still difficult to see how to utilize it effectively in the development process.

We hope that the efforts to provide a systematic catalog that has both formal and informal rationale will encourage researchers in the area to add to and improve upon the properties identified in this chapter.

For the remainder of this book, however, we will assume only the external properties defined here and try to demonstrate how systems should be built which satisfy those properties where appropriate.

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.

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 facilities 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 controlling 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 controlling 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.

11. 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

Internal Property	The Use of Software Techniques									
	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		+	+	-	++					

12. 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.

13. 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.

14. 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.

15. 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.

16. 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.

17. 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).

18. 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

Flexibility Property	The Use of Software Techniques									
	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	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.

CHAPTER 4

Software Architecture Models

4.1 Introduction

This chapter demonstrates how one can use analysis of software architectures to generate software designs that are compatible with a chosen ‘property profile’. Such a profile must be determined during requirements specification. The approach used in this chapter is to take each external and internal property, and describe (in)compatibilities between it and some interactive software architectures. Architectures developed and refined during the system and software design phases can be compatible with this profile in four ways.

1. The property is *delivered* without further developer effort.
2. The property can be *assessed* with developer effort (perhaps considerable) and skill (always extensive).
3. The property can be *addressed*, *assisted* or *measured* but cannot be assessed immediately.
4. The property is *not impacted* – the architecture does not interact with the property.

The last form of compatibility is *neutrality*, but an architecture will not be neutral with respect to all properties. Were this so, there would be no reason to use it (or the list of properties presented in this book is not complete!). The first three forms of compatibility are *support* to varying degrees. An architecture is said to be compatible with a property if it provides it automatically, supports it, or is neutral with respect to it.

During early development activity, desired properties should be selected and be allocated a priority weighting. The designer must accept these constraints during the subsequent architectural design phase. The actual process of defining and weighting properties, when coupled with the analyses developed in this chapter, can guide the design of architectures which are compatible with system requirements. These analyses are similar to those given in previous chapters, but rather than examine interactions between software techniques and quality properties in general, we restrict our attention to architectural models.

4.2 A Framework for User Interface Software Architectures

The need for architectures arises in response to complex functionality. A system's functionality is expressed as a set of capabilities ('what the system can do'). These capabilities can generally be regarded as operations on some model of an application domain, which may, for example, transform (some of) the model, or perform calculations on it.

A simple domain may only require a few operations. These could be readily understood as comprising the system's overall behavior. Typical application domains are more complex, with many operations, and comprehension must be assisted by grouping operations on the basis of similarities. These groups must be readily understood as comprising the system's overall behavior. Functionality is thus coarsened to make it manageable.

Such a grouping of the operations for an application domain is called a *functional partitioning*; this is one of the starting points for the design of a software architecture. Architectural design for software systems involves (at least) two other factors: its structure and the allocation of domain function to that structure (Kazman *et al.*, 1994). Architectural structure will be considered first.

In order to work together as a system, coarse decompositions must be recomposed by linking function groups together. The function groups must then be allocated to some architectural structure. There are two kinds of entity in this structure:

- a collection of components which represent computational entities (e.g. modules, procedures, processes or persistent data repositories);
- a representation of the connections between the computational entities, i.e. the communication and control relationships among them.

The relationships between the components must provide for efficient 'vertical' abstractions over several components, e.g. widgets composed of functions from several components.

Each allocation of function to structure should provide the designer with a different understanding of the realization of function in a software system. However, the key motivation for architectural analysis is not the creation of such understanding. Rather its purpose is to support rational choice between alternative software architectures (by comparing the corresponding allocation of function to structure in each one). Such comparisons can be guided by the probable support that each structure can deliver for desired properties for a proposed system.

4.2.1 Functional Partitioning

A *functional partitioning* is a grouping of the operations for an application domain. Here, the application domain is understood as the general one of

computer systems interacting with humans. Before proposing a functional partition for this domain, it may help to make an obvious statement:

Valid extensive conclusions about the construction of interactive systems can only be drawn if there are extensive commonalities among the systems.

Fortunately, there is extensive consensus on the adequacy of one form of functional partitioning for all interactive systems. It is based on the necessary transformations of information that flows between users and the underlying computations of interactive systems.*

In order to discuss partitioning clearly, untainted by specific features of different architectures, a standard functionality decomposition model is adopted here. The Arch/Slinky metamodel, discussed in UIMS (1992), is used with slightly modified terminology (see Section 4.5.1 for further details). The five groups of functionality used in the model are illustrated in Figure 4.1.

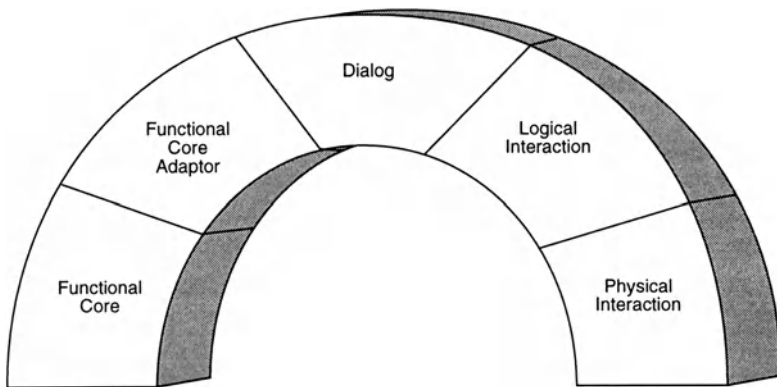


Figure 4.1 *The Arch/Slinky Metamodel.*

- **Functional Core (FC):** This group of functions implements work domain features. They are also often called the ‘application’, but that term is ambiguous (Cockton, 1987b).
- **Functional Core Adaptor (FCA):** This group of functions mediates between D and FC by providing more generic work domain concepts. They may aggregate system data into domain-oriented structures, provide a unified interface to heterogeneous FCs, perform semantic checks on data and trigger domain-initiated dialog tasks. Results from the FC are passed through to the D and onward for presentation to the user.

* Other less common partitions use phenomenology (i.e. reflecting a user’s decomposition of the system into objects), interaction structures (e.g. steps in a standard interaction cycle) or generalized interactive functions (e.g. help, customization, history) (Cockton, 1987a).

- **Dialog (D):** This group of functions mediates between domain-specific and presentation-specific functions. It controls task sequencing and context management and ensures consistency (possibly among multiple views of data).
- **Logical Interaction (LI):** This group of functions mediates between PI and D. It provides a set of logical interaction objects (sometimes called virtual objects), and presentation-specific functions to the dialog.
- **Physical Interaction (PI):** This group of functions implements the physical interaction between the user and the computer. It deals with input and output devices and is typically realized as a user-interface toolkit and/or a proprietary interface library.

This functional partitioning mirrors nicely the levels of input/output abstractions introduced in Chapter 1. It is based on an information flow *component provision strategy* (Cockton, 1991) reflecting a general observation: that the information flow starts and ends in the physical devices with which users interact and that there is a final ‘U-turn’ in an interactive system which lies deep in the *underlying functionalities* realizing the semantics of a specific work domain.*

To show the general applicability of this partitioning, a thesaurus is provided, showing how a variety of well-known architectures’ specific choices of terms relate to the terms which are used here. This is presented in Table 4.1 for a number of well-known user interface models. For each model, the functions which that architecture instantiates are given equivalents in terms of the partitioning described here. Further information about these systems may be found in Coutaz (1987), Lantz *et al.* (1987), Krasner and Pope (1988), Nigay and Coutaz (1993), Pfaff (1985) and UIMS (1992).

4.2.2 Understanding Functional Partitionings

Using the above classes of functions, an example decomposition for a simple *climate control system* is presented below. The system is to support control of temperature and humidity in a building. A control unit is used to set the desired temperature and humidity. The system monitors temperature and humidity and maintains the desired climate by controlling a furnace and air conditioning equipment. The control unit displays two sets of information: the actual temperature and humidity, and the desired temperature and humidity. Functional partitioning within this simple system shall be made to reflect the generic decomposition shown in Figure 4.1.

Functional core functions interface with temperature and humidity sensors, the boiler and air conditioning equipment; store the desired temper-

* ‘U-turns’ occur when input processing changes to output initiation. There are other ‘U-turns’ whenever feedback occurs (e.g. cursor tracking as lexical feedback for logical devices).

Table 4.1 *User Interface Functional Partitioning.*

Model	Components	Functional Equivalents
Seeheim	Appl. Interface Model	FCA
	Dialog Control	D
	Presentation	LI + PI
Seattle	Application	FC
	Workstation Manager	D
	Dialog Manager	LI
	Workstation Agent	PI
PAC-Amodeus	Functional Core	FC
	Interface with FC	FCA
	Dialog Controller	D
	Pres. Techniques Comp.	LI
	Low-level Interact. Comp.	PI
Arch	Domain Specific	FC
	Domain Adapter	FCA
	Dialog	D
	Presentation	LI
	Interaction Toolkit	PI
MVC	Model	FC
	View	LI (output only)
	Controller	LI (input only)
PAC	Abstraction	FC
	Control	D
	Presentation	LI + PI

ature and humidity; maintain the desired climate; and report the actual temperature and humidity. These constitute the underlying functionality of the system and can be independent of any user interface functions. They may be supplied by the manufacturers of the control and sensor hardware.

Functional core adapter functions convert information between the formats required by the functional core and those used by the user interface. The functional core handles the desired and actual climate as four separate numbers.

All temperature values are in Fahrenheit, as the functions are written to interface with hardware components that accept and produce digital Fahrenheit values. The user interface displays and receives temperatures in

Celsius, as the controller is designed for markets where Celsius is preferred. The functional core adapter thus includes two functions for converting between Fahrenheit and Celsius. The Celsius to Fahrenheit conversion function is called by a function that implements one of the system's two abstract commands (change desired temperature, change desired humidity).

The functional core adapter also includes a function that forms status 'records' and communicates them to the user interface. Each status message contains a pair of values, a desired setting and a current value. The user interface displays two statuses, one for temperature and one for humidity. As the functional core's reporting functions are not event based, this status forming function polls the functional core periodically to get the current temperature and humidity.

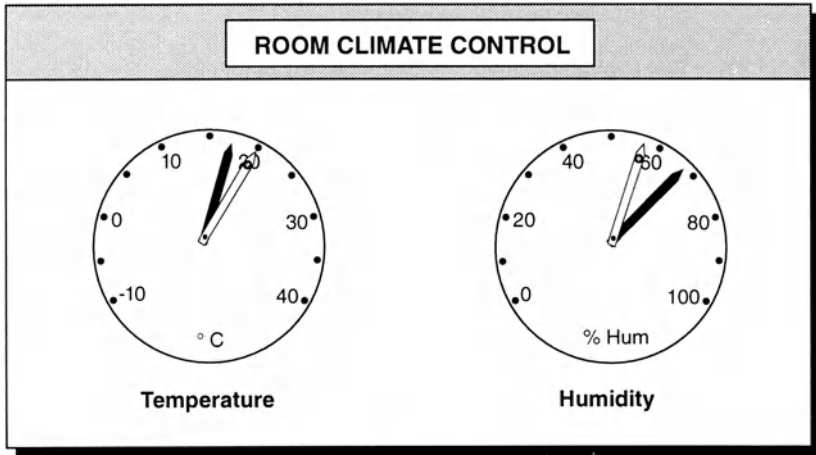


Figure 4.2 *Display for climate control.*

Dialog functions include one that implements the presentation strategy for temperature and humidity status records by converting them to parameters for the two controller-gauge widgets shown in Figure 4.2. Another function responds to a new user setting for desired temperature or humidity by calling an abstract command in the functional core adapter.

Logical interaction functions implement controller-gauge widgets. In the particular PI realization of the interface shown in Figure 4.2, each gauge has two pointers. The black pointer shows the current value and the white pointer the desired value. The white pointer has a circle near the end (the designer's intention is that this should afford manipulability). Logical interaction functions will implement these specialized widgets for a specific set of physical interaction functions. Few existing physical interaction libraries support these controller-gauge widgets. Different sets of logical interaction functions will be provided for each toolkit, window system or graphics

library that provides the physical interaction functions. However, these sets of functions would appear to be identical to any dialog client.

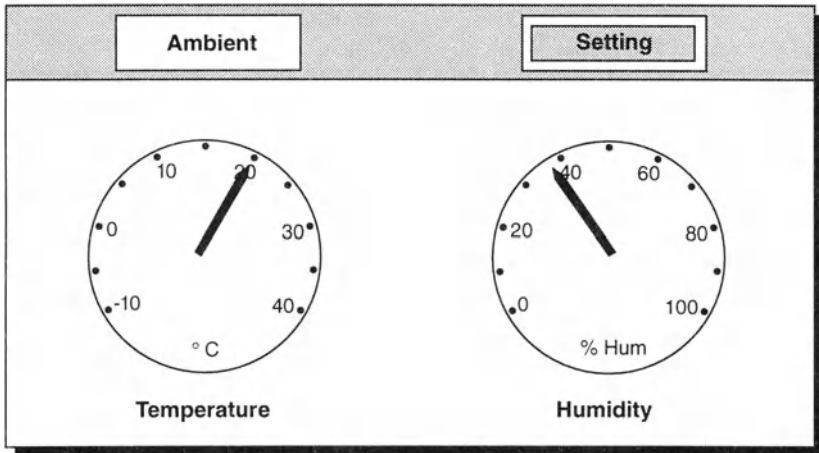


Figure 4.3 *Display with obscure button labels.*

Physical interaction functions interact directly with external peripheral devices. They display logical interaction objects as graphical primitives with requested graphical attributes. They detect user interaction with these objects. One can imagine using a device such as a mouse or a touch screen to alter the temperature.

The generic functional partitioning introduced here thus maps rather nicely onto this simple example. The partitioning can be further illustrated by the scope of changes to the user interface to the climate control system.

Changes to the way in which the data is presented to the user will affect different functional partitions. For example, Figure 4.3 shows a moded dialog with fairly obscure button labels ('ambient' means current, 'setting' means desired). For this design, some functional core adapter function(s) would have to prepare 'climate records' that pair temperature and humidity rather than actual and desired values (as in status records). Dialog functions would have to maintain two modes: in one mode, ambient temperature and humidity are to be displayed as dials; in the other state the current settings are displayed (and can be re-set). The buttons at the top of the display correspond to the two modes. A selected button is highlighted in some way to indicate the current mode. Dialog functions would respond to button presses by changing the mode. Other functions would implement behavior specific to each mode (i.e. changing the displayed information, enabling/disabling user interaction). Logical interaction functions would implement the separate control and view gauges, as well as the mode but-

tons. These functions would provide a portable interface to specific physical interaction functions.

Interestingly, this second design uses an interface which provides an explicitly modal appearance to the user, whereas the previous one was modeless. This change in the manner of operating the controls necessitates a change in the functional core adapter. It should be noted, however, that changes to the adapter could also be needed in the modeless case if the functional core provided climate records rather than actual and desired values (status records).

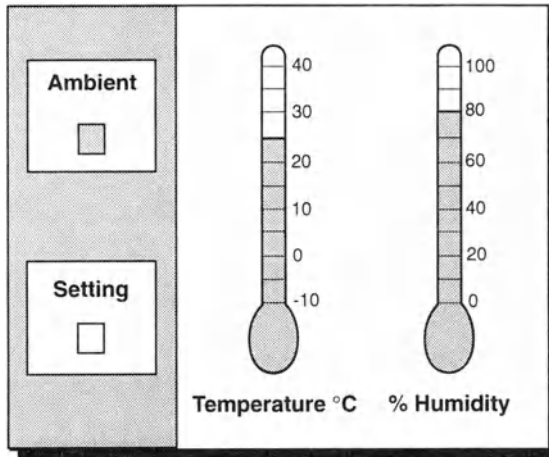


Figure 4.4 *Display with thermometer sliders instead of gauges.*

As a final example, consider the changes required to change the design in Figure 4.3 to that in Figure 4.4. Here only the logical interaction functions need to be changed (in order to render a thermometer slider rather than a dial style gauge). However, the new functions would appear to be identical to the dialog client. To do this, logical interaction functions support concepts such as 'scale' and 'selector', which they then translate into physical interaction objects, such as 'button', 'dial' or 'slider' (Figure 4.2 requires a concept like 'double-scale').

4.3 Architecture and External Properties

It is important to determine which functional partitions need to be considered when attempting to satisfy an external property. This will enable the designer to construct a system with more predictable properties. It also improves the understanding of the interrelationships between internal and external properties, as they are affected by software architecture.

The functional partitioning given in Section 4.2.1 will be used here as

the base reference. The properties are analysed for their relationships to the five canonical partitions of an interactive system – FC, FCA, D, LI and PI – illustrated in Figure 4.1.

This and the following section address the properties that support goal and task completeness, interaction flexibility and interaction robustness. If a property is considered desirable in the context of a given design then it must be considered in this analysis. For each such property it is necessary to determine which of the functional partitions will be impacted when attempting to satisfy the property. The notion is here that the list of especially desirable properties for a system will constrain the designer to certain architectural solutions.

This leads to an important observation: there are two levels of analysis at which the various properties might impact the choice and implementation of an interaction software architecture. A property might pose requirements to the allocation of function to structure, that is, it requires a specific relationship between functional partitions. It can also pose specific requirements to the implementation of run time support for one or more functional partitions. This distinction will be exemplified below. A property is said to impact a functional partition:

- if this partition must necessarily be the focus of attention in an analysis of the system with respect to the satisfaction of the property;
- if the satisfaction of this property requires extra functions to be implemented within this partition than would be the case if the property was deemed irrelevant to the design.

The discussion of how properties impact architectures is followed – in Sections 4.5 and 4.6 – by a discussion of examples of both conceptual architectural models and more implementation-oriented architectures in this context. Finally, in Section 4.7 a specific Chiron-1-based system (the climate control system described above) is assessed with respect to support for a list of given properties.

4.3.1 Goal and Task Completeness

The architectural interactions with the principle of completeness are mediated by the internal property of *functional completeness* defined in Chapter 2. The functionality required to support adopted goals and user tasks must be established early in design, and thereafter the issue becomes one of faithful realization rather than correct determination. There are thus no direct interactions between this principle and architectural models.

4.3.2 Interaction Flexibility

The analysis below examines the impact of external properties related to interaction flexibility on functional partitionings.

Role Multiplicity

This property is most visible in the dialog component. Different roles imply different dialogs handled in various subdialog components that communicate with the same functional core, or different functional cores. The overall management of these – metadialog control – has to be handled in the dialog component. In order to properly support this property, the dialog should allow decomposition into sub-dialogs. For example, in the PAC-Amodeus conceptual software architecture (described in detail in Section 4.5.3) the dialog controller is organized as a hierarchy of PAC agents (Nigay and Coutaz, 1993). To provide for the Role Multiplicity property, a metadialog mechanism which communicates with the control part of each PAC agent is needed. This also is similar to the fusion mechanism used for implementing multi-modal systems in PAC-Amodeus by Nigay and Coutaz (1995). See also Figure 4.5.

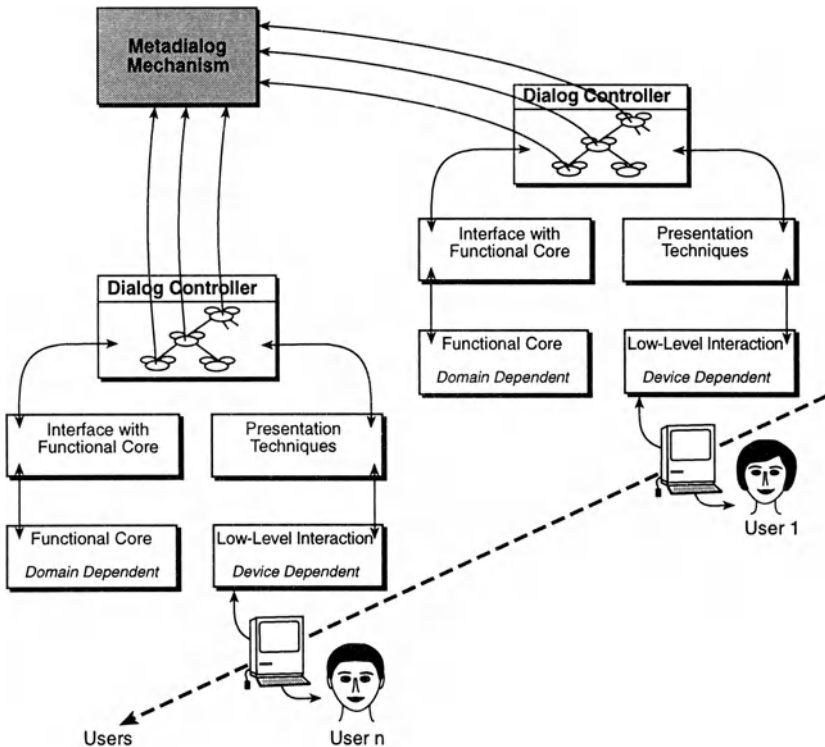


Figure 4.5 *Human role multiplicity: Applying PAC-Amodeus.*

Non-Preemptiveness

This property is most visible at the dialog level. In addition, the feedback needed to make pre-emptiveness perceivable is at the dialog level too.

For example, when trying to print a document on a Macintosh, if there is not enough memory left to print the document, a dialog box appears to indicate this fact, but the system does not allow the user to do anything else to enable printing until the dialog box is cleared. The system is pre-emptive. The Dialog defines the task sequencing, so it is clear that this component is the one that must be addressed to solve this problem.

In verifying that an interactive system is non-preemptive, the designer is greatly aided if the dialog aspects of the system are architecturally isolated. A separate dialog component will isolate the concerns about pre-emptiveness from other concerns of the system; it also isolates the point where verification of this property must be performed.

If an architecture lets dialog management be distributed – e.g. across an object space like PAC (Coutaz, 1987), MVC (Krasner and Pope, 1988) or ALV (Hill, 1992) – then it is probably easier to implement non-preemptiveness. However, it is more difficult to analyse distributed models for non-preemptiveness than if the dialog management is centralized. At the system component level, there might be different strategies for representing dialog control – some better suited for analysis than others. For example, a dialog controller built around a state transition network model provides support for the creation and analysis of non-preemptive dialogs. A dialog built with a traditional programming language (without state-transition support) would make the creation of non-preemptive dialogs more difficult.

Multithreading

In a system with multithreading the user may engage in several tasks simultaneously. This means that context management is needed when handling multithreading, and this functionality is available in the Dialog component. Hence multithreading and multitasking are strongly related to the Dialog.

If an interactive system has to support the user's desire to pursue multiple threads of interaction addressing separate tasks, then it must be able to preserve the state of any active threads of interaction. This has the following architectural implications.

- The Dialog must record the state of all threads of interaction in order to allow for arbitrary interruption and resumption of conversation threads.
- Although it is not strictly necessary to differentiate the various threads of dialog at the LI level, a reasonable architecture will reflect multiple concurrent dialogs pertaining to different user threads or tasks by means of multiple presentation schemes.

Consider the effect of these on the Dialog component: when attention

is moving from one window to another, the Dialog has to be aware of the context switch (which then results in providing feedback such as changing the cursor or highlighting the selected window).

Reachability

The reachability property concerns a user's intention in going from one state to another. As indicated in Chapter 1, states can be defined at different levels of abstraction. Therefore, from an architectural perspective, reachability affects many functional partitions. Whether the user is forward- or backward-seeking (as defined in Chapter 2), the Functional Core must allow for reachability. This could mean that sufficient history information is preserved in order to undo back to some arbitrary point in the interaction. It also means that the system avoids blind alleys – interactions which leave the FC in a state from which it is impossible to reach other states.

It may be necessary to incorporate functions that keep track of the progress of the user from one state to another in order to be able to back-track. If there are aspects of the Functional Core state which are not accessible to the user interface then changes inside the Functional Core or Functional Core Adapter are needed to permit backtracking. For example, if the FC does not provide a facility for undoing the results of previous function executions, then the FCA could keep track of changes to the state of the FC and implement undo itself. To permit run time reasoning about forward reachability, the system needs a description of the available functions at any given time. This description could be found in the Functional Core Adapter or the Functional Core.

Since analysis of reachability necessarily has to be done by looking at the dialog description, the functionality of the Dialog component is affected. For instance, a state transition diagram representation of the Dialog may indicate that the user can always return to a particular dialog state (e.g. repeated pressing of escape will return to the main menu of a menu-driven system). It is important to note, however, that this reachability constraint may not adequately reflect the user's intention, as there is no guarantee that returning to the same dialog state has had no effect on the Functional Core. The user will need to know information about the consequences of pressing an undo key in order to determine if that action will satisfy his or her intention.

An issue of forward reachability is how to indicate to the user the availability (or not) of following operations in the current interaction context. This is the property of observability which will be discussed in Section 4.3.3. The Dialog therefore must make reachability information accessible to the user.

It can be seen that reachability involves a close relationship between

functional partitions which seems to violate separation of concerns. Reachability not only has meaning in many of the identified functional partitions (FC, FCA, D and LI) but also implies that capturing the user's intention may require an evaluation within more than one functional partition at a time. For example, the discussion in the preceding paragraph for the dialog component requires an evaluation of the Functional Core and Dialog as well as the information that is communicated between them via the Functional Core Adapter. In conclusion, the present understanding of architectural separation does not localize reachability concerns. Since the property of reachability depends on functions in several partitions, no architectural modeling guarantees easy achievement.

Device Multiplicity

Dialog is independent of device and representation considerations. Therefore, concerns of device multiplicity do not extend beyond the Logical Interaction (LI) and the Physical Interaction (PI). To support device multiplicity, either the LI or the PI implementation must be able to distinguish different interaction device usages and potentially allow for their concurrent use. For instance, PI must provide time stamps so that fusion of I/O streams and time-out strategies can be handled. LI must contain mechanisms for abstracting device-dependent data into device- and representation-independent data to ensure that the Dialog is device- and representation-independent.

Some issues of device multiplicity are similar to multithreading in that two physical devices can be used concurrently.

Representation Multiplicity

This property means that a single application concept may be represented by more than one presentation object, both input and output. For example, the dials in Figures 4.3 and 4.4 represented the same underlying information, but had different appearances and different means of interaction with the user. Similarly, a new temperature or humidity setting might be specified using natural language or a command language.

The LI and PI are necessarily representation-dependent, whereas the Dialog only becomes representation-dependent when moded dialogs are selected for the representation of task methods (this is a temporal representation, but a representation nevertheless, and alternative dialog sequences will produce alternative representations).

Representation multiplicity can be achieved in a number of different ways – it does not necessarily belong to any one function. For example, one could pass a single application object from the FCA to the D, and from the D convert it into two LI objects. Or, the FC could pass a single object to the FCA, which would then split it into two objects for the D. Representation

multiplicity would not occur in the FC, since the application should remain presentation-independent, and it should not occur in the link between the LI and PI components (i.e. having the LI split a single object into two distinct presentation objects), since this should be a one-to-one mapping, for the sake of generality.

As discussed in Section 4.3.2, multiple presentation schemes can be used to effectively support multiple threads of user interaction. In this case, representation multiplicity will be achieved in the D component which is the locus for multithreadedness.

I/O Re-use

Re-use of *input* is the utilization of data previously entered in some current context. Since input re-use is a semantic function, it is ideally supported in the Functional Core Adapter and the Dialog. One of these functional partitions will gather the input (into a history buffer, for example) and enabling re-editing and re-use of it.

Re-use of *output* relies on passing the same structure of information between various partitions of the architecture. In the case of passive output re-use the LI is impacted since it has to accept that the present output representation is returned as input (for example, cut and paste features). Active output re-use is exemplified by live text, see Fraser and Krishnamurthy (1990). An example of a live text application is the ability to edit the output of a spelling checker and have the changes propagated back to the source files. The information required of the spelling checker program is simply the name of the source file(s) and the location of the (possibly) misspelled words. In order to enable active output re-use, two possible solutions present themselves:

- the LI may be modified in order to present a consistent set of input objects to the Dialog.
- the Dialog itself is modified in order to map between input objects and the requested functions.

Of these two solutions, the first appears to ensure separation of concerns – in particular that the Dialog is representation-independent.

Reconfigurability

Different types of reconfigurability may exist within several different functional partitions. Many systems allow the users to define new commands, and this belongs to the dialog level. For instance, the use of Unix shell is a way, at the dialog level, to customize the users' command line interface.

At the LI level, the style of interaction (menus or command-lines or graphical buttons) can be adjusted as may be done using the X Window resource manager. The ability to tailor displays to the user's chosen format

clearly impacts LI; for example, displaying a menu in Kanji as opposed to ASCII may require different logical interaction objects because the two representations are so different. Choosing a different set of key bindings or mouse clicks to invoke a command may impact the LI as well.

At the PI level, the use of initialization files for window managers permits tailoring the look and feel of various interaction objects, such as windows, icons, and pointers. Reconfigurability at the PI level is also available on platforms that support easy addition and substitution of input and output devices (e.g. the Apple Desk Top Bus (Apple, 1991)). Such plug-and-play capabilities interfere with software architecture since critical software support is needed in addition to hardware features.

Lastly, reconfigurability can in some cases be provided by FCA functions, for example to support a 'training wheel' approach as in Carroll and Carrithers (1984) where novice users have restricted access to a system's functionality. Such capabilities would be used by a system administrator rather than by an end-user.

Reconfigurability is defined as the user's ability to adjust I/O forms. This kind of flexibility should not be provided in the functional core because the FC deals with the domain-dependent functions of the interactive system. Changes and flexibility at this level modify the functionality of the system and belongs under the heading of the internal property modifiability in Section 4.4.

Adaptivity

Self-adaptivity is an action on the part of the system to better provide services to the user. Some examples of self-adaptivity are caching information for quicker retrieval, and automatic creation of user profiles. These types of adaptivity exist in the Functional Core, Functional Core Adapter and Dialog.

Migratability

Migratability allows transfer of control between the user and the system for performing some set of tasks. This is closely related to the Dialog component which controls the dialog and manages the sequencing of tasks. Hence the Dialog must be constructed with an ability to switch between various agents performing tasks. Furthermore, the Functional Core (or the Functional Core Adapter) must be able to initiate tasks that are 'migrated' to the system from the user.

As an example, an expert system can allow the user to transfer some decision-making responsibilities to the system and the system can decide when its decision-making process requires further user guidance. The diagnosis and correction of errors in a factory control system may have similar transfer capability between the user/operator and the system.

4.3.3 Interaction Robustness

Robustness is concerned with those features of the interaction that support successful achievement and assessment of the goals. Interaction robustness is divided into the sub-properties discussed in the following paragraphs.

Observability

A system which supports observability allows the user to evaluate its internal state without modifying it. Browsability, on the other hand, allows the user to *explore* the current internal state. This functionality is basically handled within D. However the more detailed exploration of an internal state may require (or use) knowledge from the Functional Core. For example, in the World Wide Web, browsing through the Internet will often cause files to be transferred from remote sites to the user's home site.

Insistence

Insistence deals with the the effect of a communication act. Insistence varies according to the output representation and device (e.g. audio versus graphics). Insistence will ideally be handled by the PI or LI – these will provide control over the properties of each presentation object.

However, in some cases this will affect the dialog component. Consider, for instance, a text editor which provides multiple windows to view the same document. If the user modifies the document, and wants to quit without saving, a 'Quit without saving?' dialog box will appear. This dialog box has to be acknowledged by the user before going on. If there were a single window, this insistence could be handled by the logical interaction component. In the multiple view case, however, insistence would require that *every* window should be locked under these circumstances. The user must then deal with the application query before proceeding. This insistence could be reinforced by beeps when the user attempts other actions. As the presence of multiple windows is known only by the dialog component, it will have to implement that insistence.

Insistence at the PI level is supported by hardware features such as lights on keyboards (e.g. to indicate 'Caps Lock' on). To exploit these features, LI functions must give full access to them, but they are generally abstracted away from the actual rendering device in favour of device independence.

Deviation Tolerance

As noted in Chapter 2, the system should not only help the user recover from errors but also prevent or discourage errors from occurring. However, analysing a system to discover potential error situations usually involves functionality within all the components. An example of this is seen in Dix *et al.* (1993), pages 292–5. A particular word-processor automatically saves

when the program terminates, and the user can set a flag to override this function. However, this default can be overridden by setting a (temporary) flag which allows the user to exit the program without saving the text. Of course, if this flag is accidentally set, the system enters a 'dangerous state' where the user might lose important work. Detecting such 'dangerous states' requires an understanding of the semantics of the system (ideally located in the Functional Core Adapter, with necessary support from the Functional Core). This can be seen in the example, where one needs to know that exiting without saving is a 'dangerous' thing to do. Discovering what user actions can cause these states requires an analysis of the Dialog. Finally, finding whether these sequences are easy to perform accidentally requires an examination at the Physical and Logical Interaction Levels. In the example cited one keyboard design (using function keys) gave no errors at all, but a slightly different design meant that the most common exit sequence could, by a minor typing slip, lead to the dangerous state.

Deviation tolerance for certain input errors can be supported at the LI level. For example, input can be smoothed when digitizing curves. More general provision of deviation tolerance support is found in LI modifications for users with motor difficulties; Apple Macintosh computers provide several facilities (Apple, 1991) for users with special needs (e.g. 'sticky keys', 'slow keys'). This reduces the need for speed and co-ordination of mouse buttons and keyboard modifiers. The capabilities could be regarded as supporting reconfiguration, but the aim is to remove sources of errors.

Predictability

Predictability of an interactive system means that the user knowledge of the interaction history is sufficient to determine the result of the future interaction. It deals with the user's ability to determine the effect of operations on the system. It is a user-dependent concept and is not primarily influenced by the software architecture of the interface.

Honesty

Honesty is the ability of the user interface to provide the user with an observable and informative account of the state changes effected by operations. It is a manifestation of the relation between the internal and external states of the user interface. It is defined in the Dialog which handles the mapping function between the Functional Core and the Logical and Physical Interaction. But honesty can only be obtained if the LI contains the right functionality, i.e. widgets that can produce renderings with sufficient information for the user.

Access control

Access control may affect the Functional Core or the Dialog component or the Logical Interaction component. The component affected is usually the one that manages the pieces of data to which access must be controlled. For instance, write access to a file in a Unix system is handled in the Functional Core, because files are objects of the functional core. Similarly, in a multi-user graphical editor, the locking of graphical objects can be handled in the Dialog component. Finally, in a shared editor, the temporary blocking for user interaction (where the cursor is 'locked') is relevant to the Logical Interaction component.

Pace tolerance

Pace tolerance refers to the temporal properties of the user's interaction with the Functional Core. Examples of pace tolerance are type-ahead (where the user is temporally ahead of the system), and time-outs, where the user is behind the system. Pace tolerance can affect any of the components FC, FCA, D, LI, or PI. For example, type-ahead is implemented by the Physical Interaction component. Video games are instances of applications for which pace tolerance properties are dictated by the Functional Core.

Pace tolerance also exists from the system's point of view. For example many systems provide feedback about partial completion of activities (such as fetching or copying large files).

Temporal properties cannot be demonstrated in any existing software architecture. Multi-agent approaches (as in the PAC-Amodeus model, see Section 4.5.3) can, however, provide partial feedback before the completion of a user command. This is seen as being a good way of achieving pace tolerance.

Tables 4.2 and 4.3 summarize the impact of interaction robustness properties on functional partitionings and show how well each property corresponds to a single functional partitions or adjacent partitions.

Reachability and Pace Tolerance are pervasive. Consideration of these properties cannot be restricted to a few functional partitions, but certain aspects of each property can be localized to each of the functional partitions introduced. Developers should therefore still be able to focus on one partition at a time when considering these properties.

However for the effectively pervasive properties of reconfigurability, I/O re-use and deviation tolerance, only reconfigurability can be factored into features specific to each functional partition. For I/O re-use and deviation tolerance, there will be close coupling between the source of the re-use input or output, or the source of the deviation, and its handling at a more abstract level of interaction. The value of architectural analysis here is its highlighting of two problem areas for software design. Advance knowledge

Table 4.2 *Interaction Flexibility vs Functional Partitioning*

Flexibility Property	Partitions				
Role Multiplicity			D		
Non-Preemptiveness			D		
Multithreading/Multitasking			D	LI	
Reachability	FC	FCA	D	LI	PI
Device Multiplicity				LI	PI
Representation Multiplicity		FCA	D	LI	PI
I/O Re-use		FCA	D	LI	
Reconfigurability		(FCA)	D	LI	PI
Adaptivity	FC	FCA	D		
Migratability	FC	FCA	D		

Table 4.3 *Interaction Robustness vs Functional Partitioning*

Robustness Property	Partitions				
Observability	FC		D		
Insistence			D	LI	PI
Deviation Tolerance	FC	FCA	D	LI	
Predictability					
Honesty			D	LI	
Access Control	FC		D	LI	
Pace Tolerance	FC	FCA	D	LI	PI

of such difficulties can improve development. On the other hand, the architectural analysis using the functional partitions offers no support for the creation of predictable systems.

Several properties interact with three or four of the functional partitions. A certain pattern seems to emerge here: properties such as adaptivity and migratability are mainly tied to semantic features of the system as expressed in the FC and FCA partitions; others – representation multiplicity, reconfigurability, and insistence – are tied to more representation-linked features found in the PI and LI partitions. Developers can focus their attentions on different ‘coherent’ regions of a software architecture when considering these properties. Interestingly, support for the representation-linked properties seems to be more common in real systems than support for the semantic properties. The former are given adequate to good support from

existing tools and materials (see Chapter 5), but general support for the latter has been developed almost exclusively for research systems.

Access control does not fit into the distinction between representation-linked and more semantic properties as it affects both the FC and the LI partition.

Other representational properties are more focused, such as multi-threading, device multiplicity, and honesty which are supported by two adjacent functional partitions. Observability would also have a tight architectural focus were it not for examples like Internet applications which require wide-area network (WAN) access when providing observability. Human role multiplicity and non-preemptiveness are the most focused properties. Both can be addressed by dialog functions alone.

The properties thus vary in their architectural specialization and the functional partitions vary in their influence. The relevance of dialog functions is striking. Only device multiplicity and predictability are not related to D (for very different reasons). PI functions have a limited role because many properties are supported by higher-level software processes.

There are no significant differences between the extents of influence of the other three functional partitions. However, FCA functions play a greater role in the provision of interaction flexibility than do FC ones, whereas the reverse is true for the provision of interaction robustness.

4.4 Architecture and Internal Properties

The properties described in this section are properties that are not apparent at the external level. However, the choice of a particular architecture can impact these properties. The influence which the choice of different architectures can have is discussed in the following paragraphs.

Development Efficiency

The existence of an architectural design implies that some thought has been given to software engineering considerations. One of them is efficiency in development of the actual system. Development efficiency is enhanced by a variety of means such as the ability to partition work into manageable pieces. Architectural structures which promote the division of a system's functionality into coherent classes aids in the partitioning and allotment of work.

But equally important are the ways in which these partitions are interconnected. Currently, the systems which most strongly support development efficiency are those which allow a developer to consider a 'vertical slice' of system functionality as one (large) component. Consider, for example, the File Selection widget as it appears in several current systems. Adoption of this widget speeds development not only because it provides

an encapsulation of a commonly used functionality – navigating around the file system and choosing files or directories – but also because it cuts across functional barriers. The File Selection Box contains a Functional Core (the file system), Dialog (when one selects the ‘Filter’ button, the set of displayed files is updated), and a Presentation (the buttons, sliders, labels and lists which comprise the representation of the widget).

Development efficiency benefits from the ability to *bridge* functional partitions. This, of course, conflicts with separation of concerns, which enhances re-usability – a bridge, or vertical slice, unites what would otherwise be separate functions. Thus if, for example, the developers wanted to re-use a dialog from a previous implementation, then a layered approach would be more appropriate.

To summarize, development efficiency interacts little with the capabilities of specific functional partitions. Instead, it is more impacted by the overall quality of the separation of concerns and the available ways of composing and encapsulating the functional partitions.

Modifiability and Maintainability

The introduction of reference software architectures to the user interface field was motivated by the desire to guarantee modifiability and maintainability of the software. Modifiability and maintainability offer similar design challenges. Modifiability is both supported by, and constrained by, the goals of architecture used. That is, if logically separate functionality is kept physically separate in its architectural realization then the independent modification of those separate functions is supported by this architecture. However, modifications which cut across those functions are not supported by the architecture. The more clearly defined, well motivated, and properly separated architectural components there are, the more modifiable and maintainable the resulting software will be. This separation is achieved through two mechanisms:

- separation of concerns: keeping distinct functional partitions in distinct software components;
- indirection: creating virtual interfaces, such as the logical interaction component, which buffer one component from the implementation details of another component.

To give a concrete example: when modifying a system’s dialog the software engineer should not have to worry about the effects of this change on the functional (FC, FCA) or presentation (LI, PI) components, as discussed by Kazman *et al.* (1994). If a new device is added for input or a new representation is needed, one need not change the Dialog component. Furthermore, if one wants to move from one interaction toolkit to another, one should not have to change the Dialog simply because the

attribute names of the interaction objects are slightly different. An architecture which separates these concerns properly supports the modifiability and maintainability of its software.

Thus, *indirection* may be identified as an architectural mechanism that promotes some internal properties. Another architectural mechanism that supports modifiability is the ability to homogeneously decompose a functional partition. In the PAC-Amodeus architecture (Section 4.5.3), the dialog component is refined in terms of a hierarchy of PAC agents.

Portability

Portability is a special case of modifiability. Traditional portability techniques include isolating ‘volatile’ components into a library, thus localizing the places that may require changes. Portability can thus be supported at an architectural level by separating volatile functional partitions into distinct architectural components. This was one of the main motivations of the Arch/Slinky model.

To give an example, portability across user interface toolkits is simply modifiability with respect to the PI and LI components. Separation of the PI and LI supports portability because under this model the Dialog component of any system ported would be re-usable. By way of contrast, a strict multi-agent approach might have made the porting task much harder because the PI and LI functionality is distributed across the system.

In summary, this internal property is more impacted by pervasive architectural qualities than by a specific functional partitioning. Portability depends on a specialized notion of separation of concerns, such that any volatile functional partitions are isolated from those functions that implement the underlying domain semantics.

Evaluability

This principle is architecture-neutral, since the choice of architecture does not affect how easy it will be to measure the quality of the final system. The concept of an architecture is still important since it may be easier to isolate, in a well-structured system, where evaluation has to be done.

Run Time Efficiency

Given that systems do not have unlimited resources, the efficient usage of system resources is always important. If data have to move through many layers and have to be transformed at each layer then this naturally leads to inefficiency at run time. Thus a layered architecture does not lend itself well to maximizing run time efficiency. As with development efficiency, one needs the ability to take ‘vertical slices’ of system functionality if high performance is a priority.

For example, it has often been commented that the Seeheim model of user interface software needed to provide a special mechanism to support semantic feedback. This mechanism bridged layers (functional partitions) in the Seeheim model, as indicated by the small, unlabeled box in Figure 4.6, taken from Pfaff (1985).

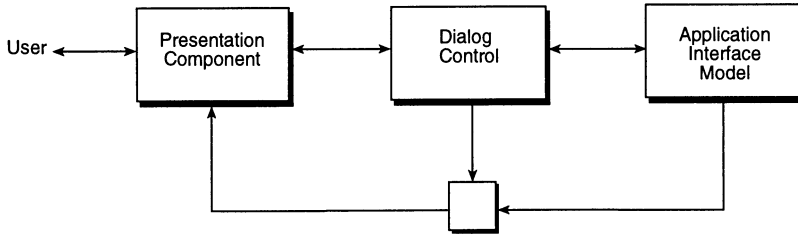


Figure 4.6 *The Seeheim Model.*

Semantic feedback occurs when the presentation is changed in real time according to the *semantics* of a user's input. For example, in the Macintosh desktop, if one drags a file over a folder icon or the dustbin icon, these icons will be highlighted. If, however, one drags a file over another file, the file will not be highlighted. The highlighting of the dustbin or the folder are examples of semantic feedback, indicating that these objects are potential locations for the file being dropped. Semantic feedback involves the use of application (FC) information – knowing the purpose of each screen icon – in the presentation (PI) component.

If one implements each functional partition of the Arch/Slinky model as a separate layer, creating interfaces between each of the five components – FC, FCA, D, LI and PI – then each time the user moves the mouse, an event could pass from LI to FC, and back again.

The frequency of this event may vary. The PI may pass events to the LI each time the mouse moves by a single pixel, but an efficient LI would only pass enter and leave events to D each time the mouse left the bounds of one object and entered another selectable one (the LI can be configured to only pass events for selectable objects, as in PRESENTER as introduced by Took (1990)). Depending on the configurability of the LI, no communication between D and FCA may be required during dragging. With weak abstractions in the LI, and no way for the FCA to indicate selectable objects to the LI (via D), the location information would have to cross eight interfaces. The current pointer position would need to travel from PI to FC (so that the type of the object currently underneath the pointer could be checked) and back again (to actually highlight the covered object, if necessary). This layering is clearly sub-optimal.

If such advanced LIs and FCAs cannot be easily implemented, one would prefer instead to be able to take a vertical slice of functionality, from FC to

PI, and bundle this functionality together so that as little as possible unnecessary work is done, and as few as possible layer boundaries are crossed.

It sometimes occurs that a system will bridge layers for a small subset of functions, but not for the entire system. This could occur in the following ways.

- An entire Arch could be embedded within a single functional partition. This type of layer bridging is found in any sophisticated widget which provides access to some Functional Core services. An example of this is the File Selection Box widget referred to above.
- A functional partition may directly call a service in another functional partition to which it is not adjacent. An example of this is the layer bridging in the Seeheim model (the ‘bypass channel’), as illustrated in Figure 4.6.

In summary, pervasive architectural qualities rather than a specific functional partitioning impact this internal property. Support for ‘vertical slice’ capabilities as a composition and encapsulation mechanism may improve run time efficiency. But also key architectural decisions on the allocation of function to structure have a direct bearing on this property. The division of labour between LI and PI partitions has a critical effect on the bandwidth of the interfaces between these components. For run time efficiency, we want to minimize the bandwidth requirements among a system’s components, particularly if those components are physically separated (say, communicating across a network). The right allocation of function to structure will minimize the events that functions in the D partition must deal with.

Functional Completeness

As defined in Chapter 3, a system is functionally complete if the various abstract commands and state elements required to support the designed task model can be faithfully implemented. Within the scope of this chapter, the term ‘faithfully’ means that the system is efficient at run time and easy to develop. A good example is the semantic feedback issue discussed above. A strictly layered architecture would here make it difficult to implement this concept faithfully whereas a more fine-grained object oriented architecture like ALV (Hill, 1992) would allow for the necessary trade-offs between maintainability and run time efficiency.

User Interface Integratability

The ability to integrate an application into an existing environment, ensuring that the user interface is compatible with interfaces of other applications in the environment, is a difficult task. It involves achieving a look and feel which is consistent with existing applications. This is typically achieved via user interface toolkits for LI/PI functional partitions. User Interface

Integratability can be achieved through consistent interaction techniques – such as menus, form-filling, drag-and-drop, etc. – and consistent dialogs (e.g. using consistent syntax).

This internal property is the only one with a narrow architectural focus in the presentation components (LI and PI). In part this reflects the rarity of extensible support for semantic external properties such as adaptivity and migratability in current systems.

4.4.1 Internal Properties and Functional Partitions

There are three architectural principles which significantly impact internal properties.

Separation of concerns – or the overall principle of functional partitioning, which varies across the external properties, and brings specific benefits for internal properties where indirection (e.g. virtual interfaces such as LI) is provided.

Division of labor – or the overall allocation of function to structure, which should preserve the overall quality of the functional partitions.

Composition and encapsulation – or the basic provision of structure, which should allow different ways of combining and re-using functional partitions, for example, the ‘vertical slices’ mentioned above.

For internal properties, the structure of an architecture, and its relation to the functional partitioning rather than the functional partitions themselves, has more profound implications than for external properties. The remainder of this chapter examines these structural issues in more depth.

4.5 Conceptual Architectural Models

The previous two sections introduced the notion of analysing or choosing a systems architecture on the basis of our quality properties. The discussion was based on a generic partitioning of functions for interactive software systems in order to be as broadly applicable as possible.

This and the following section analyse the allocation of function to structure and take a closer look at two conceptual architectural models and two more implementation-oriented software architectures. This aims to reinforce the understanding of the issues introduced above. It also serves to lead up to a concrete example of architectural analysis in Section 4.7.

4.5.1 The Arch Model of Interactive Systems

The Arch model of interactive systems is in UIMS (1992) defined as a layered structure. The functional core of the system to be designed and the UI

toolkit provided by a given implementation environment form its two endpoints between which three additional component layers are interspersed. The model makes direct use of the functional groupings adopted above (with small differences in nomenclature from Figure 4.1). It comprises:

Domain-Specific Component – which controls, manipulates and retrieves domain data and performs other domain-related functions.

Domain-Adapter Component – a mediation component between the Dialog and the Domain-Specific Components. Domain-related tasks required for human operation of the system, but not available in the Domain-Specific Component, are implemented here. The Domain-Adapter Component triggers domain-initiated dialog tasks, reorganizes domain data (e.g. collects data items in a list), and detects and reports semantic errors.

Dialog Component – which has responsibility for task-level sequencing, both for the user and for the portion of the application domain sequencing that depends upon the user; for providing multiple view consistency; and for mapping back and forth between domain-specific formalisms and user-interface-specific formalisms.

Presentation Component – a mediation, or buffer, component between the Dialog and the Interaction Toolkit Components that provides a set of toolkit-independent objects for use by the Dialog Component (e.g. a ‘selector’ object that can be implemented in the toolkit using either a menu or radio buttons). Decisions about the representation of media objects are made in the Presentation Component.

Interaction Toolkit Component – which implements the physical interaction with the end-user (via hardware and software).

Domain objects are used by both the Domain-Specific and the Domain-Adapter Components, but instances of these objects are created by the two components for different purposes. In the Domain-Specific Component, Domain objects employ domain data and operations to provide functionality not associated directly with the user interface. In the Domain-Adapter Component, domain data and operations are used to implement operations on domain data that are associated with the user interface. For example, one domain-specific operation of a database management system (DBMS) would retrieve a set of employee names and salaries by gender from a database. Iterative review of the list to display parts of succeeding records might need to be done in a Domain-Adapter Component. Here the Domain-Adapter Component would supplement the functionality of the Domain-Specific Component by providing a service related to the presentation of information.

Presentation objects are interaction objects that control user interactions but are toolkit-independent. Presentation objects include descriptions of

data to be presented to the user and events to be generated by the user. The medium used in the presentation or event generation is not defined. An example of a Presentation object for use with the list of employees and salaries is 'tabular, labeled, two-column data with single-entry selection'.

Interaction objects are specially designed instances of media-specific methods for interacting with the user. Interaction objects are supplied by the Interaction Toolkit software and may be primitive (e.g. graphics and keyboard device drivers) or complex. An Interaction Object corresponding to the Presentation object cited in the paragraph above is a dual bank of radio buttons (which allows the user to select an employee with a particular salary from the 'male' column or the 'female' column).

4.5.2 Migration and Branching in the Arch/Slinky metamodel

The above description of Arch could give two misleading impressions:

- functions cannot migrate from their 'logical partition' in an architecture;
- there is no branching into multiple partitions.

Neither is true. Architectural analysis only arises because migration is possible. Otherwise, only one allocation of function to structure would be possible. However, the levels of abstraction given in Chapter 1 do indicate a 'logical partitioning' of functionality in addition to providing important analysis guidance. Too great a departure from the 'spirit' of a functional partition normally reveals itself as a negative impact on internal and/or external properties.

For example, the logical level of interaction requires comprehensive device-independence in its abstraction over physical devices. Development efficiency requires that this is delivered in as compact a form as possible. Also, the dialog level requires clear isolation of *interaction points* in order to support walkthroughs for the assessment of external properties (but especially those determining interaction flexibility). Lastly, the functional level requires capabilities that are compatible with (the user's model of) the work domain. Incompatibility has a negative impact on role multiplicity, predictability, honesty, observability, customizability, and migratability.

In other words: concepts defined at the functional level of abstraction should logically be implemented in the Functional Core (Adapter) component, concepts defined at the dialog level of abstraction should be implemented in the Dialog component, etc. However, the correspondence is not strict; for instance, a strategy of semantic delegation or semantic repair will implement some concepts at the functional level of abstraction in a dialog component, see Bass and Coutaz (1991).

Another migration factor is that sophistication in UIMS's dialog languages is required since dialog functionality (sequencing, constraint maintenance, context management) is inherently complex. Consequently, many

dialog languages are Turing complete, or are extensions of existing Turing complete languages. One can therefore, in principle, do almost anything in the dialog partition of many UIMSs. This does *not*, however, mean that everything in the system is inherently dialog and should be dealt with in one big chunk.

In summary, not only are there good reasons for migration in architectures, but there are many requirements that, once satisfied, make it impossible to prevent migration. Thus when it was stated in Sections 4.3 and 4.4 that a property interacts with some functional partition, this expressed the fact that the property affects the portion of a system which is logically concerned with the partition, irrespective of how or where that functionality is implemented. The analyses thus identify the groups of functions that logically impact properties, leaving aside the problem of what it is *possible to implement*.

Migration is dealt with in the Slinky generalization of the Arch Model as found in UIMS (1992). The coupling of functionalities in the layers of the Arch model described above was designed to minimize the effects of future changes in the interaction toolkit, the user interface dialog or the application domain. Dissimilar functions were assigned to separate components in order to allow the modification of one type of functionality with minimal impact on other components in the system. However, a model derived to minimize the effects of changing technology may have an adverse effect on the speed of the run time system. A single model cannot satisfy conflicting criteria – i.e. different sets of critical quality properties.

The Slinky metamodel provides a set of Arch models, as opposed to one particular model. The Slinky metaphor was chosen since function groups can migrate through the arch in the way that the coils of a Slinky toy (a large and long spring) may distribute themselves in many different ways throughout their arch.

To clarify the concept of shifting functionalities, consider an example where a function in a Domain-Specific Component was later implemented in an Interaction Toolkit. The Unix file system was originally considered a specific application domain, with file operations such as ‘open’ and ‘delete’. When the interaction toolkit became more sophisticated, a file selection widget was included in the toolkit, thus shifting the functionality from one end of the model architecture to the other. In one sense, the file selection widget can be regarded as a simple string widget at the LI level, but this ignores the extensive file system functions behind it, which can and do make changes to the functional state of the system (such as current drive and directory).

A second complication with architectural models is the need to provide for sub-partitioning of functional classes on the basis of some system context – known as *branching*. This can be added to the Arch/Slinky model as indicated in Figure 4.7 for example.

The causes of branching are varied, but include the following.

Technology – variations in target environments may require multiple adapter partitions, e.g. multiple logical interaction partitions for multiple toolkits at the physical interaction level, or multiple FCA partitions to provide unified access to different kinds of databases (SQL-based, object-oriented, etc.) through the same interface.

Re-use – extensions to a system’s capabilities may be achieved by re-use of an existing component. It may be impossible, and will probably be unwise, to incorporate the new component into an existing functional partition.

Closely related applications may need separate Functional Cores, but their interaction with the user should be (almost) identical. One way of handling the addition of separate Functional Cores is to have a single Functional Core Adapter interfacing with all of the Functional Cores. Similarly, for the second factor, a single Logical Interaction component could interface with all Physical Interaction toolkits. However, as long as a Functional Core or Physical Interaction software package doesn’t interact with other packages (e.g. by sharing a limited resource like a communication channel, a graphics display device, or a locator device), then it may be preferable to accommodate *multiple* Functional Core Adapters and Logical Interaction components within a system.

Suppose a new Functional Core is added to a system that currently has one instance of each functional partition. Suppose also that the data and functionality of the new Functional Core are independent from that of the old one. In such a case the existing Functional Core Adapter should not be changed to generalize to both Functional Cores, for to do so would ruin the integrity of the existing Functional Core Adapter. In such a case a new Functional Core Adapter should be added to mediate for the new Functional Core, thus isolating the existing Functional Core Adapter from this change to the system as well as from future changes to the new Functional Core. In this way, the two Functional Core Adapters communicate independently with the Dialog Component, forming two branches which join at the Dialog Component (Figure 4.7).

In short, maintaining single instances of functional partitions may not be worth the effort. Branching of the Slinky structure supports multiple instances of functional partitions. Branching also lets a user interface development environment be functionally distributed and modularized by creating networks of components.

These examples of discretionary branching are however less common than imposed branching. In CSCW systems, it is only possible for multiple users to share the same Functional Core (Adapter).^{*} Each user requires a ‘leg’

^{*} The delivery of the WYSIWYG (what you see is what you get) property can be controlled by multiple Functional Core Adapters.

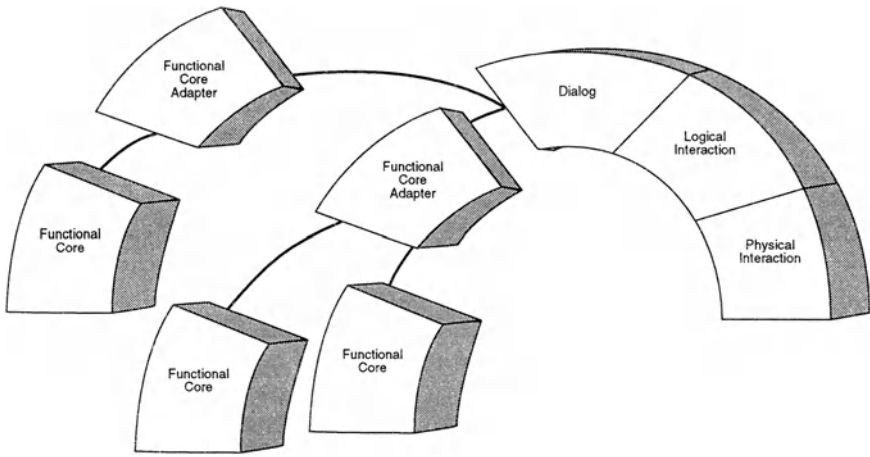


Figure 4.7 *Branching. The Arch/Slinky Metamodel with Multiple FCs and FCAs.*

of the Arch/Slinky branching off from a shared Functional Core (Adapter). This is why branching is the appropriate metaphor for multiple partition instances, since several partitions may be composed. A second example of such branching is when several Functional Cores are visualized on a single screen. This results in several Arch/Slinky structures converging into a single Physical Interaction component.

Branching is also useful for the LI/PI components. Consider an interactive system used by people with different linguistic or cultural background. The same functional core is then equipped with rather different user interfaces implemented as multiple LIs or PIs.

4.5.3 The PAC-Amodeus Conceptual Architectural Model

As alluded to in the previous subsection, branching is a problem not originally addressed in the Arch/Slinky metamodel. Another issue arises when considering recursive decomposition of interactive systems which in many cases is a useful method for designing complex systems, such as systems with highly interactive direct manipulation user interfaces. In such cases layered separation of functionality is not sufficient as the only decomposition principle. The PAC-Amodeus model (Nigay and Coutaz, 1993; Coutaz *et al.*, 1995) is defined as an Arch/Slinky-oriented extension of the original PAC model (Coutaz, 1987) addressing migration, branching and recursive decomposition.

PAC-Amodeus adopts the same components as Arch and assigns the same roles as Arch to these components. However, PAC-Amodeus goes one

step further than Arch by decomposing the Dialog component into a set of cooperative PAC agents, as illustrated in Figure 4.8.

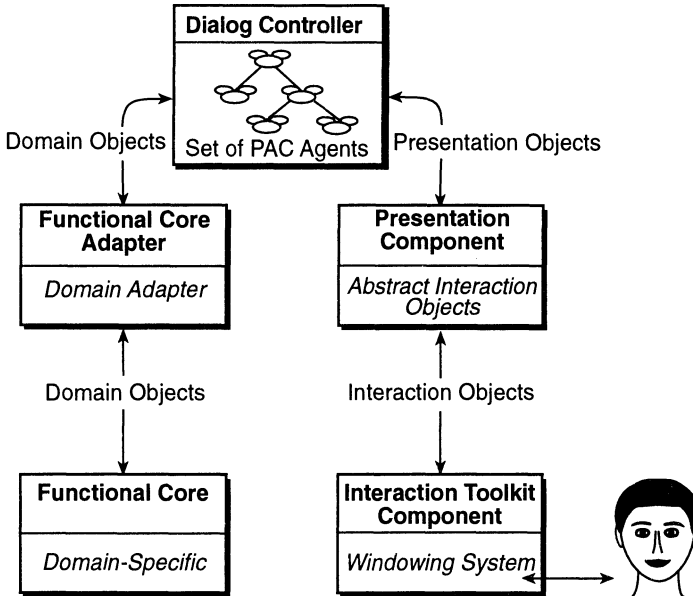


Figure 4.8 *The PAC-Amodeus Model.*

The Dialog Controller has the responsibility for task-level sequencing. Each task or goal of the user corresponds to a thread of dialog. This observation suggests the choice of a multi-agent model which distributes the state of the interaction among a collection of cooperating units. Modularity, parallelism and distribution are convenient mechanisms for supporting multi-thread dialogs. One agent or a collection of cooperating agents can be associated with each thread of the user's activity. Since each agent is able to maintain its own state, it is possible for the user (or the functional core) to suspend and resume any thread at will.

The Dialog Controller receives events both from the Functional Core, via the Functional Core Adapter, and from the user via the Presentation Component. Bridging the gap between a Functional Core Adapter and Presentation Component has some consequences. In addition to task sequencing, the Dialog Controller must perform data transformation and data mapping:

- A Functional Core Adapter and a Presentation Component are oriented in different directions. One is driven by the computational considerations of the Functional Core, the other is toolkit-dependent. In order to match

the two different styles, data must be transformed inside the Dialog Controller.

- State changes in the Functional Core Adapter must be reflected in the Presentation Component (and vice versa). Links must therefore be maintained between domain objects of the Functional Core Adapter and presentation objects in the Presentation Component. A domain object may be rendered with multiple presentation techniques. Therefore, consistency must be maintained between the multiple views of the conceptual object. Such mapping is yet another task of the Dialog Controller.

Thus, bridging the gap between the Functional Core Adapter and the Presentation Component covers task sequencing, formalism translation, and data mapping. Experience shows that these operations must be performed at multiple levels of abstraction and distributed among multiple agents.

Levels of abstraction reflect the successive operations of abstracting and concretion. Abstracting combines and transforms events coming from the presentation techniques into higher-level events for higher abstractions. Conversely, concretion decomposes and transforms high-level data into low-level data. The lowest level of the Dialog Controller is in contact with the presentation objects.

This multi-agent approach supports parallelism, distribution, multithread dialogs and iterative design. Since agents should carry task sequencing, formalism transformation, and data mapping at multiple levels of abstraction, the Dialog Controller is described at multiple grains of resolution combined with multiple facets. At one level of resolution, the Dialog Controller appears as a ‘fuzzy potato’. At the next level of description, the main agents of the interaction can be identified. In turn, these agents are recursively refined into simpler agents. This is the usual abstraction/refinement paradigm applied in software engineering.

Orthogonal to this refinement/abstraction axis, the ‘facet’ axis is introduced. An agent is described along three facets: Presentation, Abstraction, Control. These facets are used to express different but complementary and strongly coupled computational perspectives.

- The Presentation facet of an agent implements the perceivable behavior of the agent. As shown in Figure 4.8, it is related to some presentation object of the Presentation Component.
- The Abstraction implements the competence of the agent (i.e. its expertise) in an essentially media-independent way. It is the Functional Core of the agent. It maintains the abstract state of the agent. It may be related to some domain object(s) of the Functional Core Adapter. The abstraction facet of an agent provides a good mechanism for performing domain-knowledge delegation.

- The Control part of an agent is in charge of two functions: linkage of the Abstraction part of the agent to its Presentation portion and maintenance of the relationships of the agent with other agents. The linkage serves two purposes: i) formalism transformations between the Abstraction and the Presentation portions of the agent, and ii) data mapping between the abstract facet and the presentation facet. Relationships between agents may be static or dynamic. Dynamic relationships are required when agents are dynamically created/deleted. Relationship maintenance by the control part of an agent covers the communication and the synchronization mechanism between this agent and its cooperating partners.

In summary, a PAC agent could be viewed as a mini-Arch. Figure 4.9 shows how one PAC agent relates to other agents and to the surrounding world of the Dialog Controller:

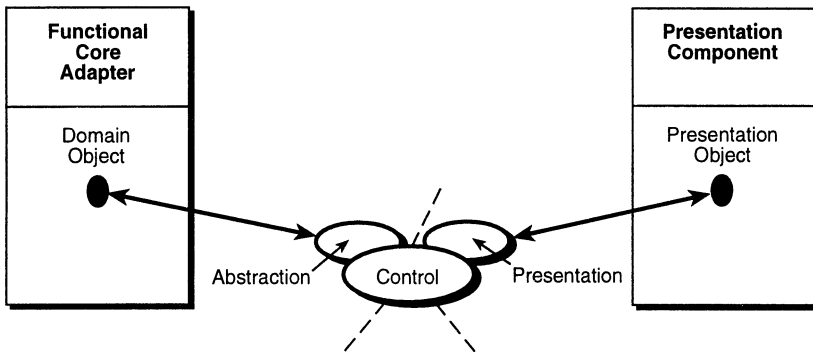


Figure 4.9 A PAC agent of the Dialog Controller.

4.6 Example Architectures

In the following examples the Serpent and Chiron-1 architectures are discussed in the light of the generic functional partitioning described above and from the viewpoint of satisfying the quality properties introduced. Research systems have been chosen for the following reasons.

- Research tools are often concerned with software architecture, and so provide enough information in their descriptions to allow architectural analysis. This is often not the case for commercial systems.
- One commercial tool should not be advocated over another in this context.

A graphical convention for representing architectural structure is introduced, see Figure 4.10. This in combination with the generic functional partitioning will ease comparisons and discussions of the architectures.

Rectangles with solid lines represent processes, or independent threads of control, ovals represent computational components which only exist within a process or within another computational component (e.g. procedures or modules), shaded rectangles represent passive data repositories (typically files), shaded ovals represent active data repositories (e.g. active databases), solid arrows represent data flow (uni- or bi-directional) and grey arrows represent control flow (also uni- or bi-directional).

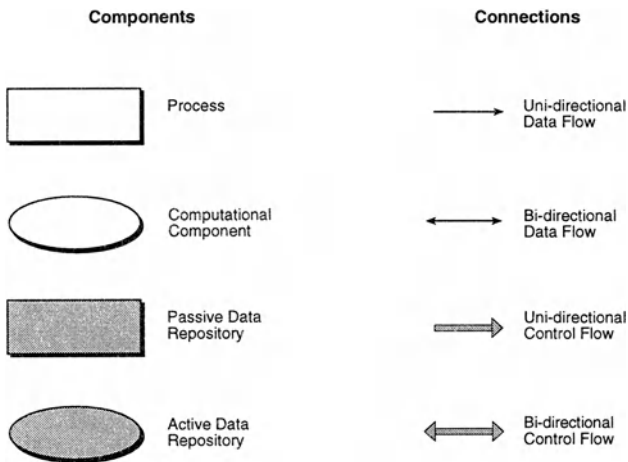


Figure 4.10 *Structural Notations.*

4.6.1 *Serpent*

Serpent identifies a dialog controller, the presentation and the application as three distinct processes in its architecture, as shown in Figure 4.11.

Application modules contain the computational semantics required for the application. Although there can theoretically be many different applications contained within a given run time instance of Serpent, there is typically only one. Presentation modules provide techniques for supporting interaction at both the logical and physical level completely independently of application semantics. Different presentation modules in a given run time instance are possible, although not typical. Given that application and presentation modules are separate, there must be a way to coordinate a given application component with a presentation component. That is the purpose of the dialog controller. The dialog component mediates the user's interaction with an application, through the control of the presentation.

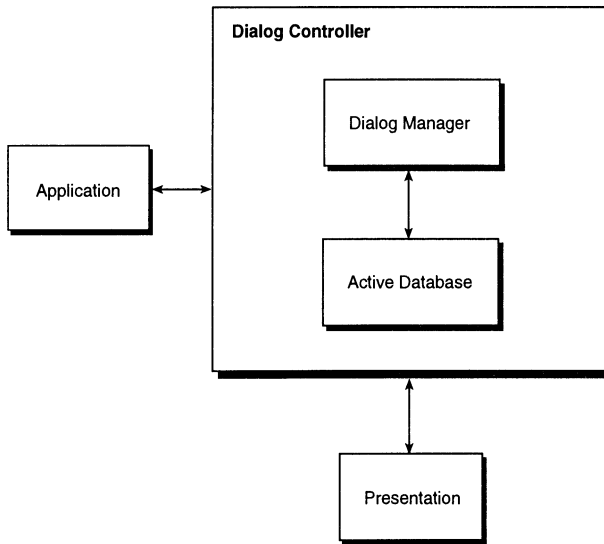


Figure 4.11 *Serpent's Architecture.*

All communication between Serpent components is mediated by constraints on shared data in the database shown in Figure 4.11. This structure is implemented as an active database; when values in the database change, they are automatically communicated to any component which is registered as being interested in the data. This global database physically resides in the same process as the dialog controller but is logically independent from all of the Serpent components. A dialog manager sits within the dialog controller process and mediates the connection between application and presentation. The dialog manager is further decomposed into a collection of view controllers – not shown in Figure 4.11 – which provide a finer grain of correspondence between application and presentation objects.

4.6.2 Analysis of Serpent

This section explains the mapping between the system-specific notion which Serpent implements and the functional partitioning used as the base of reference in this book. Figure 4.12 recasts Serpent's architecture in the common functional notation given in Section 4.2.

Several things have been changed between Figure 4.11 and Figure 4.12:

- the Active Database is represented as an active repository;
- data and control relationships are exposed;
- independent flows of control are exposed through the delineation of processes; and

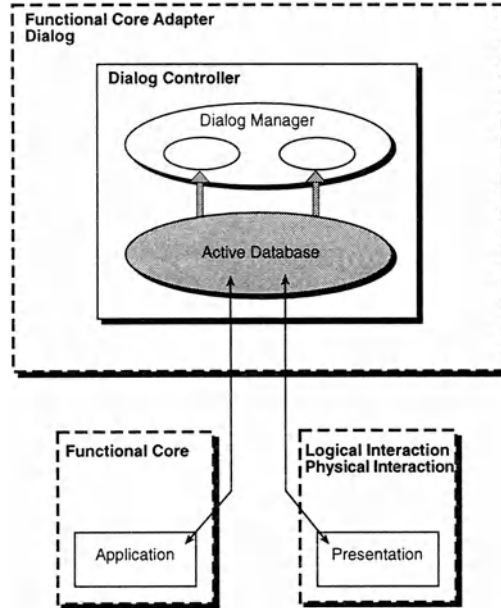


Figure 4.12 *Serpent's Architecture (annotated)*.

- a view controller hierarchy is exposed as a subdivision of the dialog manager.

Recall that, with properties associated with functional roles, and with Serpent analysed in terms of these roles, Serpent can now be assessed against a property profile. This assessment will be discussed after the analysis of another example architecture.

4.6.3 Chiron-1

Chiron-1 (Taylor and Johnson, 1993) is a User Interface Design System which was created with the goal of addressing two important software life-cycle issues: maintainability and sensitivity to environmental changes. Chiron-1's architecture, as presented by its authors, is shown in Figure 4.13.

A Chiron-1 system consists of a client and a server. The client consists of an application, which exports a number of abstract data types (ADTs) which Chiron-1 encapsulates within Dispatchers. Dispatchers communicate with Artists, which maintain abstract representations of their associated ADTs in terms of an abstract depiction library (ADL).

A Chiron-1 server consists of: a virtual window system, which translates from abstract interface depictions into concrete ones; and an instruc-

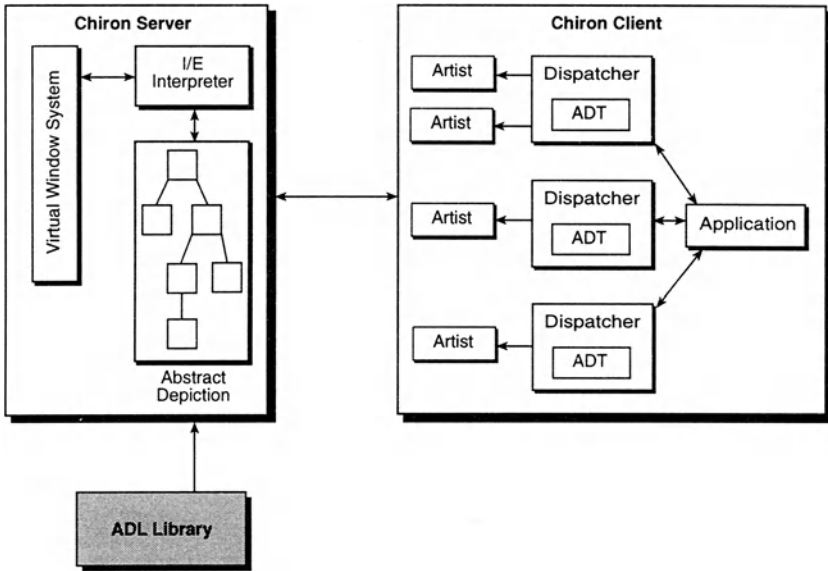


Figure 4.13 *Chiron-1's Architecture (original)*.

tion/event interpreter. It also accesses an ADL. The instruction/event interpreter responds to requests from Artists to change the abstract description and translates those requests into changes to the presentation. The server also responds to events from users and translates those back into Artist requests.

All of these components – ADL, Artists, virtual window system etc. – are specific to Chiron-1. This ‘naming problem’ makes analysis and comparison with other systems difficult. The goal in doing architectural analysis is to have a single language and a single representation for understanding architectural issues. In order to do this, Chiron-1 is re-characterized in terms of the functional roles given in Section 4.2.1. This leads to a system-independent language for talking about functionality, and the partitioning of functionality.

4.6.4 Analysis of Chiron-1

The Chiron-1 architecture clearly separates the application (functional core) from the rest of the system, as would be expected in a system which was built with the expressed goal of minimizing sensitivity to environmental changes. The functional core adapter could live in the ADTs, or in the Artists. It seems clear that the Artists contain some of the dialog since

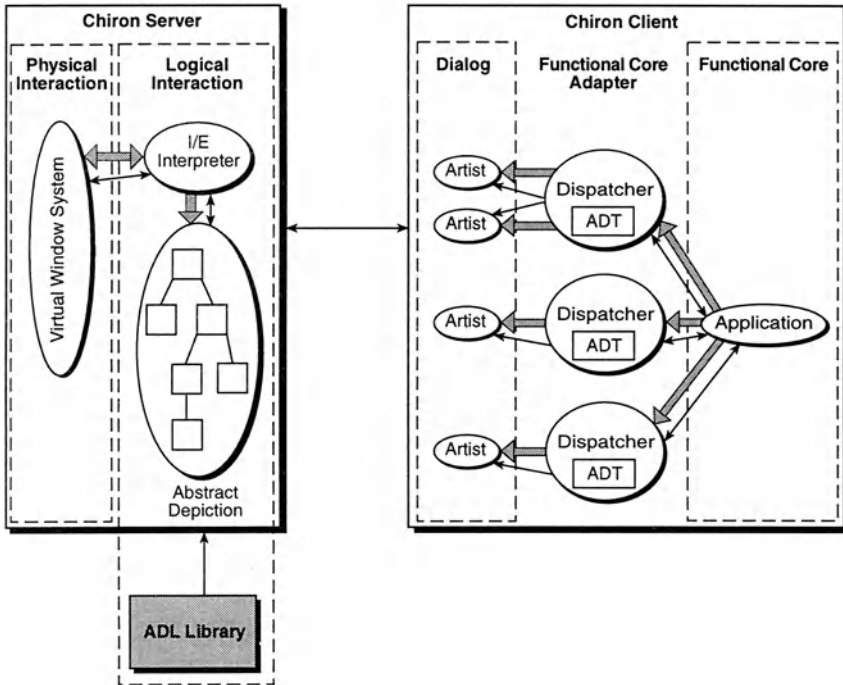


Figure 4.14 *Chiron-1's Architecture (annotated).*

they have the job of maintaining a correspondence between objects from the application domain and interface objects from the presentation domain.

However, what is less clear from Chiron-1's architectural description is where the 'state' of the dialog lives. For example, where does one put the information that the 'Paste' option in an edit menu should be greyed out unless something has previously been cut or copied? Another type of dialog issue is maintaining relationships among the interface objects. For example, when a user selects the 'Save As' option in a file menu, something in the dialog must cause a file selection box to be created.

The location of these sorts of dialog issues is explicitly addressed in Chiron-1's architectural description. These dependencies might exist in the Artists, in the ADTs or even in the Abstract Depiction Libraries. For simplicity's sake, we have provisionally annotated the architecture to show the dialog as living completely in the Artists.

The final two functional roles are clearly identified: the Physical Interaction function corresponds to Chiron-1's Virtual Window System component, and the Logical Interaction functionality is provided by the I/E

interpreter, augmented by the ADL. As a result of this characterization, the Chiron-1 architecture is provisionally annotated as shown in Figure 4.14.

In this depiction of Chiron-1 (which was adapted from the original system architecture given in Taylor and Johnson (1993)), the partitioning of functionality can be viewed in terms of the functional roles given in Section 4.2.1.

4.7 Assessing Quality Properties

The main aim in analysing the allocation of functional roles to a specific architectural structure is to support assessment of an architecture against a property profile (as long as properties have been associated with functional roles).

Before giving examples of such an assessment for both Serpent and Chiron-1, the above analysis of Chiron-1 can be cross-checked by mapping the temperature/humidity example onto Chiron-1's architecture. Performing this mapping has the further advantage of allowing assessment of properties for a specific realization of an architecture for a specific (small) computer system.

4.7.1 A Chiron-1 Architecture for a Climate Control System

The functional core of the climate control system (Section 4.2.2) – getting current or ambient temperature and humidity and setting the desired temperature and humidity – will be located in the 'Application' component, labeled FC in Figure 4.14. The functional core adapter – responsible for conversion and bundling/unbundling of information – will be located in the Chiron-1 'ADT Dispatcher' components, labeled FCA.

The dialog functionality for the climate control system – reporting user requests to the application, displaying application information to the user and switching between setting mode and ambient temperature mode – should be located in the Artists, as indicated by the D label surrounding the Artists.

The presentation functionality in Chiron-1 is all located in the Chiron-1 server. The logical interaction portion, which translates between generic presentation objects and particular window-system specific objects, is located in Chiron-1's Abstract Depictions and I/E interpreter, as indicated by the LI label. The physical interaction component, which would display the controller-gauge widgets to the user and receive input from the user, is located in Chiron-1's Virtual Window System, as indicated by the PI label.

The functions of the climate control system are now partitioned and mapped onto the structural components of a realized user interface architecture. This allows establishment of the software ramifications of user-

oriented properties. If it were desirable, for example, to guarantee a particular property of a user interface – e.g. that a user can always return to a previous state no matter what their current state is – it would be necessary to assess how this property would affect the software of the system being developed. One key concern is to know which portions of the system need to be changed to better support a property. This is done for a few sample properties below.

4.7.2 Assessment of Properties

Representation Multiplicity

Consider, for example, the property of representation multiplicity. As stated in Section 4.3.2, representation multiplicity can be manifested in a number of different ways – it does not necessarily belong to any one function or one software component. Two possibilities suggested were:

- the FC could pass a single object to the FCA, which would then split it into two objects for the D;
- a single application object could be passed from the FC to the D, and from the D converted into two LI objects.

Consider how this would be manifested in a system. A system such as Chiron-1 could implement the first possibility. A single application object, created in the FC, would then be packaged by Chiron-1 as an ADT (in the FCA). This ADT would be split into two realizations and passed, via a Dispatcher to two distinct Artists (dialog components).

In another system, one might choose to split an application object at a different point. For example, the Serpent User Interface Management System described in Bass *et al.* (1990) divides a system into FC, D/FCA and PI/LI components, as shown in Figure 4.12.

In Serpent, the FCA and D functionality are undifferentiated, and so there is only one way to achieve representation multiplicity: a single application object is passed to the FCA/D component, where it is split into two. Each of these two objects in Serpent's Active Database would then correspond to distinct LI/PI objects.

Insistence

The same kind of analysis of a software architecture can be made with respect to other properties as, for instance, the property of insistence (Section 4.3.3). Insistence, it should be remembered, deals with the duration and period of the effect of a communication act. Consider a visual interaction object which is to have a blinking aspect.

In order to achieve this function in Chiron-1, the designer has a choice between three possibilities. If the underlying medium provides this function,

it will be located in the PI – Chiron-1's Virtual Window System – and he or she needs only choose the appropriate objects and attributes. If the PI does not provide it, one would ideally want to simulate this functionality in the LI. The LI provides a consistent set of interaction objects to the Dialog, smoothing over or hiding the idiosyncratic differences of individual toolkits.

In order to achieve insistence then, the LI would have to simulate an insistent visual object by 'blinking' it – alternating its background and foreground colors. In the Chiron-1 system, this functionality would have to be located in the Abstract Depiction library. If, however, the LI could not or did not support such functionality, then it would have to be supported in the dialog component – in the Artists, in the case of Chiron-1.

In Serpent roughly the same situation is found. Insistence could be supported in Serpent's Dialog Controller, or in its Presentation (which does not architecturally differentiate Logical Interaction and Physical Interaction).

Modifiability

Modifications to the user interface are easiest to perform when the fewest modules must be changed. Let us examine one specific modification: 'grey out menu items that are currently not accessible'. This modification requires knowledge of currently valid menu choices and this knowledge is contained in the Dialog. Both Serpent and Chiron-1 isolate Dialog within their architectures and so this type of modification is assisted by both of them.

User Interface Integratability

Suppose that a new system written using Serpent or Chiron-1 is to be introduced into an existing environment. Since the details of the presentation come from the Physical Interaction partition, and since this partition is localized in Chiron-1 it can be seen that this architecture assists User Interface Integratability. Since in Serpent the Logical Interaction and the Physical Interaction partitions are bundled together, Serpent is less effective in this than Chiron-1.

4.8 Conclusion

Chapter 3 introduced a set of software phenomena that interacted with properties under the broad heading of software techniques. It identified software architecture as the phenomenon that interacted most with our properties. In this chapter, we have examined these interactions. To do this, we have described software architectures in more detail, and have presented a common framework for architectural analysis. The functional groupings that were adopted in this framework can be related to our prop-

erties and thus identify where attention can be focused when attempting to satisfy a property. This helps with the choice of an architectural model that is optimal for a prioritized list of required properties. Even so, choice between different published software architectures is difficult. There are always different ways of applying a given architectural model to a given design problem, especially as their descriptions in the literature are not sufficiently detailed. Architectural analysis would become more straightforward if architectural models were accompanied by guidelines on how to apply them. Furthermore, they could be further motivated with reference to those properties that they supported well.

Despite these difficulties, we have shown how properties can guide the choice of a system software architecture. Such guidance is most straightforward when properties are associated with a single functional grouping. For example, reachability and properties that involve mappings by dialog functions such as observability are most easily checked for in architectures such as Serpent where the dialog is explicitly represented and localized. Analysis of these properties is far less straightforward for architectures such as PAC-Amodeus or Chiron-1 where the dialog state is distributed across multiple agents or artists. Conversely, multi-threading, representation multiplicity and device multiplicity are more easily handled within multi-agent approaches like PAC-Amodeus where each thread of dialog is realized by a separate sub-hierarchy of agents. For both sets of examples, analysis is simplified by our prior association of properties with functional groupings. Once an architecture has been related to these functional groupings, a property profile can easily be established for an architectural model, which in turn simplifies the analysis of specific instances of a software architecture.

CHAPTER 5

Tools and Materials

5.1 Introduction

An interactive system is seen by different people from different points of view. The system user is concerned with external properties, such as those that influence task coverage, flexibility and robustness during system use. The developer is often more concerned with those internal properties which address such things as the costs and reliability of development throughout the entire development life cycle.

The subject of this chapter is the interactions between properties and software techniques that are methodological in nature. Chapter 3 identified interactions with these software techniques as the next most significant after those with software architecture (this formed the subject of Chapter 4).

Effective and efficient use of methodological techniques is unlikely without tool support. For example, quality procedures, which validate that a system meets its requirements, are likely to fail without extensive support from tools and materials. Since these interactions are mediated by the use of various tools and materials, this chapter examines how these influence properties.

The wide range of tools and materials used for examples in this chapter (e.g. the use of automatic code generation techniques and hypertext approaches to requirements structuring) extends the subset of software techniques identified in Chapter 3. However, no attempt is made to cover all possible software techniques (e.g. change control tools and protocols).

5.1.1 Definitions

In the context of this chapter, *materials* are defined broadly as anything that someone in a development role produces for a specific project. In addition to specifications, implemented code, and evaluation reports, materials may include design documents, system administrator as well as well as user documentation, on-line help and tutorials, and various training aids. Even marketing strategies can be regarded as materials given the above definition. However, the definition unfortunately excludes any re-usable code that was not produced for a specific project (e.g. interaction toolkits). The definition could clearly be improved, but in order to keep it simple, the

comment may be added that anything that could be produced by a development role for a specific project can be alternatively provided by re-using existing materials.

Materials mark the boundaries of software development phases. They are used to pass information between phases (evaluation materials can reflect information back into a phase). In contrast, *tools* embody the activities that carry a project forward. Materials produced with tools in one phase are used either in subsequent phases to generate further materials, or within the same phase to refine other materials within it. For example, design specifications can be transformed by model-based tools into executable code, but they can also be analysed with evaluation tools in order to produce evaluation reports that guide further refinement of the design specification.

5.1.2 Potential Scope of this Chapter

The potential scope of this chapter is very large. For example, five broad categories of material can be identified.

1. **Requirements materials** specify the requirements for an interactive system. Requirements specialists select external and internal properties and allocate weights to them.
2. **Specifications and Design materials** provide a detailed description of the interactive system.
3. **Coded modules** implement the components of an interactive system. Implementers transform design materials into these coded modules, except where existing materials can be re-used.
4. **Working system**, i.e. the coded modules, bound together and executable, resulting in an interactive system performing useful work, and containing state information about a user's current interactive tasks.
5. **Evaluation reports** describe the strengths and weaknesses of a working system, partly expressed in terms of the properties identified in Chapters 2 and 3.

Development roles from Chapter 1 appear above (e.g. Requirements specialists, Implementers) and below. Recall that multiple roles can be filled by the same person, and that a single person may fill multiple roles.

The usual input/output relationship in any one phase is defined by the application of a transformation to the input in order to create the output (e.g. transform a task method into a dialog sequence). However, other forms of input/output relationship may be needed during system development. For example, in the case of mapping from requirements to specifications, a solely transformational approach cannot handle the pervasive nature of

'non-functional' requirements, because there is no simple mapping of requirements onto design features. In this case, a *checking* relationship predominates, designs being checked against requirements. In all cases, tools may be used to move between different categories of material. Five categories of such tools can be identified.

1. **Requirement tools** are used by requirements specialists to formulate requirements.
2. **Specification tools** are used by system designers to produce specification materials that describe intended solutions.
3. **Construction tools** are used by implementers to transform specification materials into coded modules.
4. **Execution tools** are used by system administrators to assemble and bind modules into interactive systems.
5. **Evaluation tools** are used by validators in evaluating interactive systems by exercising and measuring various usability aspects.

The above lists of categories of materials and tools cover a potential range that is so large as to make this chapter's analysis unmanageable. The working group therefore made several pragmatic decisions that restrict the scope of analysis. The scope is also restricted by a number of logical considerations that do exclude many tools and materials from the discussion.

5.1.3 Restricting the scope of this chapter

The main pragmatic restrictions result from an assumption that the most relevant materials for an analysis of interactions with properties are those that describe the final system, evaluate these descriptions, specify properties for the final system, or remain in the final system (i.e. as coded modules or resources that are referenced during execution). It is also appropriate to consider detailed architectures as materials (they are extensions of the architectural models considered in Chapter 4). Given the restrictions on 'relevant' materials, 'relevant' tools generate such materials (this extends transitively to all 'ancestor' tools and materials in a 'generation pipeline').

These pragmatic restrictions exclude tools that support standard methods from Usability Engineering (Nielsen, 1993), as these evaluate the final system rather than a description of it. This is a little late for our purposes. Such tools (e.g. Hix and Hartson, 1994) are placed outside the scope of this chapter, even though user testing is required to establish the satisfaction of all user-dependent properties (e.g. honesty), as well as properties given a low (even no) priority in an initial property profile.

A further pragmatic restriction is that we ignore hardware materials, even though pace tolerance is generally determined by system response time (which in turn is improved by high-performance processors, accelerator boards etc.).

Other restrictions placed on the tools and materials considered below can be supported by argument. Primarily, many tools and materials in routine use do not interact with properties. For example, general purpose text editors can be used to create specifications, but provide no support for internal or external properties. At best, such tools provide some support for development efficiency, but experiences with CASE over the last decade suggest that even this is open to question. Actual usage can often be an act of faith, and some tools used in good faith are actually detrimental to the achievement of high-quality interactive systems.

Further logical restrictions arise from the development life cycle. Tools and materials for the early phases of development can be ignored, since properties are not selected nor are weightings allocated until late in the requirements specification phase. Neither should the phases following the system test be considered, since properties should have been established long before system installation. The scope of this chapter may therefore be logically confined to the phases from the start of system design until the end of system testing (but with usability evaluation tools already excluded from the scope). These remaining phases may be collected into the following three groups.

Specification – the phases of system design, software design and module design.

Construction – the phases of coding, module tests and integration tests.

Evaluation – the system test phase.

The fine-grained phases of Chapter 1 are now, therefore, replaced by coarse-grained groups of phases. These are used to organise the analysis in the same way as functional partitions did in Chapter 4. However, there is one class of tools and one development practice that clearly cut across the above coarse groups. These must be considered before examining interaction in detail.

Model-based tools cut across coarse development phases. These address specification, construction and evaluation by automatically generating large parts of the final system. Clearly, however, internal properties may be assisted, as these tools integrate and manage activities that span most of the development life cycle. For example, ADEPT (Johnson *et al.*, 1995) begins with task analysis and user modeling, expresses requirements as a task model and a user model, and then generates intermediate models (design specifications) from which code is generated. Tools such as DON (Kim and Foley, 1993) and TRIDENT (Vanderdonckt and Bodart, 1993) also evaluate models for various qualities. Such tools can be accommodated by treating them as a family of implicit tools. Each implicit tool is used to address a separate coarse development phase.

Prototyping (see Section 1.3.1) cuts across coarse development phases. At

least three uses for a prototype are possible once an iterative development is halted.

1. The prototype and evaluation reports become materials for *requirements specialists*, who transform them into formal requirements that reflect the whole prototyping experience.
2. *System designers* draft formal specifications that capture key features of the prototype at all levels of abstraction – here prototypes are materials that specifications are checked back against.
3. The prototype contributes coded modules for use by an *implementer* of the final system (who will add modules for new functionality and/or user interface capabilities).

The third use arises with evolutionary prototypes, while the first two arise with rapid prototypes (as defined in Section 1.3.1).

Rapid prototypes can be constructed using paper and pencil, or with tools such as HyperCard (Goodman, 1993) and Director (Macromind, 1990). They can be evaluated, for example, to provide some confidence about interaction flexibility or robustness. Tools with explicit high-level configuration languages could even support proof of some properties. Rapid prototypes are thus *reference* materials against which requirements or specifications can be checked. They are strictly part of the early development phases and thus rapid prototyping tools are outside this chapter's scope.

When using rapid prototypes, the speed at which they can be produced limits evaluation to the external properties defined for the system since software quality standards (i.e. internal properties) must necessarily be relaxed in creating them. However, when developing evolutionary prototypes, high software quality standards must be maintained at all times since the prototypes develop into the final system for which high standards are wanted. It follows from this that evolutionary prototypes will usually be developed using commercially available construction tools.

For evolutionary prototypes, the relevant properties, tools and materials are no different to those for final systems that have been developed without prototyping. Such commercially available user interface development products include:

- UIMX from Visual Edge
- InterfaceBuilder from NeXT
- Prototyper from SmethersBarnes (SmethersBarnes, 1990)
- Visual Basic from Microsoft
- PowerBuilder from PowerSoft
- XFaceMaker from Non-Standard Logics.

In summary, this chapter will focus, for a mixture of pragmatic and logical reasons, on materials and tools that are used or produced during

specification and construction. Model-based tools can be analysed first for their specification support, and then for construction. Evolutionary prototypes can be regarded as being no different to any other final system for our analysis. The ten categories of tools and materials that could provide the scope for this chapter have thus been reduced to a manageable four (there are no evaluation tools or materials to consider).

Detailed discussion of relevant interactions begins by considering first those interactions between properties and tools/materials within the specification and construction phases of development (Sections 5.2 and 5.3 respectively). Examples will be drawn from a wide range of existing tools and materials. This analysis of interactions is then extended in two ways: first by examining three well-established tools across a representative range of properties (Section 5.4); then by presenting current practice in using such tools at four representative development sites (Section 5.5).

5.2 Specification Tools and Materials

The time consuming task of specification spans system design, software design and module design. Recall that the properties which must be satisfied during these design phases will have been selected and allocated weighting during earlier development.

5.2.1 Flexibility Properties

Consider first the need for flexible planning of task execution. This involves the properties of reachability, non-preemptiveness and multi-threading.

Reachability can be proved when using some notations, especially ones for dialog abstractions such as transition networks. The RAPID prototyping tool (Wasserman, 1985) configures dialogs using state transition networks, letting reachability be at least assessed at the dialog level of abstraction. Proof is obstructed by RAPID's traversal semantics – there are side-effects on transition conditions, input consumption and time-outs (Cockton, 1985). For tools with cleaner transition conditions and traversal functions, path algebras (Alty, 1984) can be used. These can compute the transitive closure of state transition graphs, and thus be used to assess reachability and representation multiplicity (for input, through multiple dialog structures). Such tools are still only present in research environments and applicable only to moderately sized systems (Alty and Ritchie, 1985).

Transition networks and similar dialog abstractions also support assessment or proof (given a formalization) of *non-preemptiveness*. Unfortunately, these abstractions effectively obstruct the *multi-threading* property. This is because networks can only represent interleaving of processes by having a path for every possible trace through the process complex. Interleaving two network-specified dialogs with m and n states respectively

Table 5.1 *Specification Phase Interactions between Tools/Materials and Interaction Flexibility Properties*

Property	Interaction	Comment
Reachability	Prove	Most straightforward with 'clean' dialog abstractions
Non-preemptiveness	Assess	By inspecting specifications that support proofs of reachability
	Deliver	By using dialog abstractions with process constructs
Multi-threading	Obstruct	By using any sequential dialog abstraction
	Address	By using dialog abstractions with process constructs
Device Multiplicity, I/O Re-use and Human Role Multiplicity	None	Dependent on construction tools/materials
Representation Multiplicity	Assess	Same relationships as observability with constraints, view controllers, model-based tools and cognitive walkthrough
Reconfigurability, Adaptivity and Migratability	None	Dependent on construction tools/materials

requires a combined interleaved network with $m \times n$ states, whereas production systems require only $m + n$ rules to interleave two rule sets of m and n rules (Hill, 1987).

This problem with networks can be overcome quite simply by directly addressing the *multi-threading* property by adding process constructs (England, 1988; Jacob, 1986), but care must be taken with the underlying *schedulers* that distribute events to different dialog networks. Often the schedulers will not preserve the desired properties of good concurrent processes. Multithreading and non-preemptiveness are thus best supported by using integrated specification constructs that address them directly, (e.g. Statecharts – Harel, 1988).

Table 5.1 summarizes interactions between properties and tools or materials (T/M). Overall, tool support for flexibility properties is biased towards properties that can be proved or directly provided in some form. Only reachability can be easily proved (and this will largely be at the dialog level).

Multi-threading and non-preemptiveness can be delivered by appropriate control constructs, which again are best suited to specialization for the dialog level of abstraction.

Support for interaction flexibility during specification is very limited, and largely restricted to flexible planning of task execution. This is because many of the capabilities required for interaction flexibility must be provided by construction and execution tools. Specification tools tend to lag behind construction and execution tools, but there are clear opportunities for better support here.

Classifying Interactions

Several new terms are introduced in the second column of Table 5.1, and these will be explained before returning to the main analysis. Each form of interaction is defined by a set of activities in which developers must engage in order to exploit the interaction (with the exception of *None* and *Obstruct*, see below). To define each form of interaction, we must first identify the defining activities. For the forms of interaction in Table 5.1, these are.

Specialization – developers use a basic knowledge of a property to instantiate a construct (e.g. instantiating an interaction specification for a pull-down menu using an appropriate* construct such as a transition network).

Formalization – developers must use extensive knowledge of a property to express it as a formal predicate.

Proof Discharge – developers must use formalization, specialization (to produce a specification) and extensive skills at following proof procedures to establish that a property holds for the specification.

Inspection – developers must use extensive knowledge of a property to inspect specifications (which ideally should be formed from instantiations of appropriate constructs).

Given these activities, four interactions can be defined as follows.

Delivery – involves none of the four activities, and yet a positive interaction still results (by use of appropriate tool or re-use of materials).

Proof – requires formalization then specialization then proof discharge;

Addressing – requires only specialization.

Assessing – requires inspection after specialization.

* An *appropriate* construct is implicitly defined as one that requires only a basic knowledge of a property to instantiate it. Once properly instantiated, in the sense that the instantiation is well formed, the property is delivered. The developer has to do nothing other than to 'fill in the blanks', although this may involve quite complex expressions.

The table also includes an *obstruction* interaction. This can be the opposite of either assessing or proof, as the degree of obstruction may be assessed by inspection or established by a proof procedure. Here, developers must engage in activities to establish that a property *cannot* be (fully) supported.

When there are no positive or negative interactions between a specific tool or material and a property, then the tool or material is said to be *neutral* with respect to this property. In this case, no combination of development activities using the tool or material could exploit a property or demonstrate that it was obstructed. 'None' in the table indicates that all the tools and materials that were considered were neutral with respect to the property in question.

There are clear advantages in defining interactions in terms of required development activities. Firstly, it reveals some forms of interaction as being specific to a development phase. Thus tools that support *proof* of properties are used during specification. Furthermore, some interactions during early phases create further tasks for later phases. Properties that are proved or addressed during specification must be *preserved* during construction (for addressing, this can be achieved by construction tools that address properties to the same standard as specification tools).

A second advantage of defining interactions in terms of required activities is that it reveals differences in entailed developer effort for each form of interaction. Delivery requires no further developer effort beyond use of appropriate tools or re-use of materials (as, for example, use of a true functional programming language will deliver *referential transparency* for all programs); proof requires extensive developer effort and skill – someone must produce design specifications, someone must formalize the property and someone must do the proof; addressing requires some developer effort – someone must form relevant parts of a specification by instantiating an appropriate construct (e.g. processes are appropriate for forming instances of non-modal dialog boxes); assessing requires developer effort and good human factors skills – someone must produce design specifications and someone must inspect them.

A third advantage of defining interactions in terms of required activities is that it reveals differences in likely attainment of each form of interaction. Consider, for example, the activities involved in proof. Only the specialization activity is straightforward (for developers who can use specification tools). In contrast, formalization of properties can be very frustrating. It was not completed for any informal property in Chapter 2, despite several efforts. Many apparently acceptable predicates turn out to be too weak or too strong, i.e. they admit or exclude design features that do not or do support the property that they should formalize. Proof discharge is also a risky enterprise. The ability to prove a property depends on the notations used. Thus proof procedures for some properties are well understood when established constructs such as transition networks and context-free gram-

mars are used. However, Chapter 4 identified no proof interactions between architectural models and properties, for although *architectural description languages* are being developed (Garlan and Shaw, 1993; Luckham *et al.*, 1995); this work is still at an early stage.

In summary, forms of interaction between tools/materials and properties have a straightforward definition that involve one or more development activities in a particular order. As long as the simple basis of their definition is remembered, they support valuable analyses that identify what developers must do and when they must do it. This is especially valuable when identifying properties that can be ignored from an early stage in development, cannot be attended to during long periods of construction, require little developer effort, require much developer effort, can be reliably exploited or incur risks of failure. Having presented the basis for such judgements, we can resume the main analysis.

5.2.2 Robustness Properties

Flexibility properties are quite general and can thus often be addressed by general formal computing constructs, at least for flexible planning of task execution. In contrast, robustness properties require more constructs specific to interactive systems. As will be seen, general constructs can be used for pace tolerance, but several properties can only be assessed by using walkthrough techniques. Overall, several constructs are often required to address a robustness property comprehensively. This is particularly the case with deviation tolerance. But specific architectural support is needed for observability, and this robustness property is examined first.

Observability is defined as the possible rendering (at the logical level of abstraction) of relevant state (at the functional level of abstraction). Architectural models described in Chapter 4 support separation into different levels of abstraction. So do UIMSs that link functional components to interaction components via a dialog component.

When such architectures are embodied in specification tools, then *link constructs* are required to address observability. A link construct is any specification construct (or software entity) that forms connections between separate architectural components. A range of link constructs is mentioned in one of the example site reports (Section 5.5.2 below).

Dialog functions that use specialized link constructs address observability. This holds because information must be rendered to become observable, and dialog functions are responsible for initiating such renderings (by transferring selected information from the functional core adapter to selected logical interaction functions). Hence link constructs such as SERPENT's View Controllers (Bass *et al.*, 1990) address observability by encapsulating design decisions on when and how to render information.

Developers need not be aware of link constructs in model-based user

interface development tools such as ITS (Wiecha *et al.*, 1990), Humanoid (Szekely *et al.*, 1993), and UIDE (Sukaviraya *et al.*, 1993). They generate appearance and behavior from higher-level models, addressing observability by (semi-)automatically linking functional level models to presentation models.

Lastly, link constructs can also partially deliver *honesty* as long as the system is *pace tolerant* with minimal delays between functional state and corresponding display updates, since a value on the display should always accurately reflect its underlying value. Automatically created links can promote honesty if pace tolerance conditions are met.

In the absence of architectural support, *observability* and related robustness properties (i.e. *insistence*, *honesty*) can be assessed in specifications, primarily by combining design descriptions with walkthrough procedures. For example, *Cognitive Walkthrough* (Wharton *et al.*, 1994) combines task descriptions in any format chosen by developers with a simple set of questions. Four questions are asked at any interaction point:

1. Will the user try to achieve the right effect?
2. Will the user notice that the correct action is available?
3. Will the user associate the correct action with the effect they are trying to achieve?
4. If the correct action is performed, will the user see that progress is being made towards solution of the task?

Two of these questions address two stages of Norman's Seven Stage Model of Human-Computer Interaction. In this model, users cycle through command execution and result evaluation (Norman, 1986). Before entering commands, users must work out what to enter. Norman calls this stage 'action specification', and question 2 addresses it. Question 4 addresses all three of Norman's three result evaluation stages (perception, interpretation and evaluation).

There are clear similarities between some robustness properties and Cognitive Walkthrough questions.

Predictability is covered by question 3;

Honesty is covered by question 4;

Insistence is covered by question 2 (and question 4 is implied by honesty);

Observability: both questions 2 and 4 are implied by insistence.

Cognitive Walkthrough thus supports assessment of four robustness properties. It is, however, a paper-based tool, although specification tools could easily be extended to step designers through each of the four questions at each interaction point.

Other assessment methods focus on a single robustness property. For example, predictions of learnability made by Cognitive Complexity Theory

(CCT) let *predictability* be assessed (predictable systems are more consistent than unpredictable ones and thus require fewer rules than a user model for an inconsistent system), although the effectiveness of CCT is disputed (Knowles, 1988).

CCT focuses on a single robustness property, and it also requires a complete system model for CCT at some level of abstraction (usually the dialog level). In contrast, for Cognitive Walkthrough, task descriptions may only give partial coverage at mixed levels of abstraction. In short, designers can walkthrough what they *imagine* the system to be, rather than what some specification says *it is*. There are trade-offs between developer effort and comprehensiveness here. Only partial assessment can be expected unless task descriptions are formally derived from a complete system specification.

Considering other robustness properties during specification, *pace tolerance* can be assessed, and perhaps even delivered, by general computing constructs. Real time specification languages can be combined with real time scheduling algorithms to establish that a response to an event will occur within a given time (Burns, 1994). For example, rate-monotonic scheduling algorithms (Sha and Sathaye, 1993) can be used to establish pace tolerance, although they currently ignore system overheads. They also largely address hardware issues such as bus protocols. Their applicability to current interactive systems is limited, but progress with such approaches could be relevant for pace tolerance. With control over processing time, predictability in the form of response time stability would also be addressed.

More specific tools can focus on interface details that impact pace tolerance, for example, messages that are displayed and removed under system control. Users must be given time to read these. Algorithms for calculating the necessary duration of a message exist (Bevan, 1983), and thus the times could be specified. A focused tool could address by making such calculations.

The authors are aware of no interactions between specification tools or materials and the robustness property of *access control*.

The remaining robustness property, *deviation tolerance*, must be addressed by a wide range of constructs during specification. For example, input validation constructs, which are increasingly found in user interface builders, only address the error detection aspect of deviation tolerance. The property must be further addressed by constructs for error prevention and error recovery. For example, error prevention constructs can be found in screen layout tools that include constructs for preventing users from performing inappropriate commands (e.g. the presentation of an undesirable command can be visually distinguished at the logical interaction level).

Error prevention can be given more general support by specification languages with command pre-conditions. The earliest UIMS work here was by Mark Green (1985). Pre-conditions over states at the functional level

are used in the UIDE environment by Sukaviraya *et al.* (1993), where they support automatic generation of various user support features such as intelligent help (Sukaviraya *et al.*, 1992). More extensive pre-conditions, which support error detection, error prevention and error recovery, are found in the NUF notation, a specification notation for the functional level (Cockton *et al.*, 1995). This has four types of pre-conditions for abstract commands: availability, prevented failure, automated recovery and mixed-initiative recovery.* Availability pre-conditions prevent users from initiating a command (typically by greying it out and making it unselectable). Prevented failure pre-conditions are the simplest form of error detection, which merely specify an error state from which no further recovery is possible. Automated recovery pre-conditions specify an error state from which automated recovery is possible. Mixed-initiative recovery pre-conditions specify an error state from which recovery is possible, but only with user involvement.

Consider, for example, a document editor which offers a command to the user to save the document as a file to be named by the user as part of the command interaction (sometimes known as a 'Save As' command). When no documents are being edited, this command is made unavailable (availability condition is a non-empty set of open documents). The editor must also prevent the file save failing because an unacceptable file name is given (prevented failure condition is presence of special characters, incorrect <prefix>.<suffix> format, or name too long), as well as negotiating mixed-initiative recovery for the case where the named file already exists (that is, the error detection condition) – to prevent unintentional over-writing of contents (error recovery takes the form of a yes/no/cancel question). Also, where a programming environment offers a command to run the program being developed it can automate error recovery where the current version of the source code needs recompilation before execution (the error detection condition is 'source changed since last compilation', and error recovery takes the form of recompilation).

Current model-based tools do not provide such recovery mechanisms. In Humanoid (Szekely *et al.*, 1993), for example, side effects must be used to provide for error recovery. In UIDE (Sukaviraya *et al.*, 1993), a generated dialog model may have to be extended by hand to include recovery. Support for the deviation tolerance property may therefore be diminished between specification and construction.

Overall, there are few sufficiently general constructs that address robustness properties. Developers must thus wait until construction phases where re-usable library materials can provide specialized assistance for specific

* Mixed-initiative recovery involves both the user and the system in deciding whether to quit without saving.

Table 5.2 *Summary of Specification Interactions between Tools/Materials (T/M) and Interaction Robustness Properties*

Property	Interaction	Comment
Observability	Address	Constraints/View Controllers
	Assess	Model-Based User Interface Generators Cognitive Walkthrough Questions 2 and 4
Insistence	Assess	Cognitive Walkthrough Questions 2 and 4
Honesty	Assess	Cognitive Walkthrough Question 4 Temporal aspects assessed in conjunction with both observability and response time conformance
Predictability	Assess	Cognitive Complexity Theory (but effectiveness disputed (Knowles, 1988)) Response Time Stability assessed along with pace tolerance Cognitive Walkthrough Question 3
Access Control	None	Dependent on construction materials
Pace Tolerance	Deliver	Real time scheduling algorithms (potentially)
Deviation Tolerance	Address	Partial support from UI management tools/builders with input validation construct
	Address	Pre-conditions as used in NUF (Cockton <i>et al.</i> , 1995) and Model-Based User Interface Generators
	Assess	Cognitive Walkthrough can establish effects of errors

design features. Interactions between tools/materials and robustness properties are summarized in Table 5.2.

Robustness properties are given more extensive, but less effective, support than flexibility properties by tools and materials. Only observability, pace tolerance and restricted forms of honesty and deviation tolerance can be addressed. Otherwise, assessment is the best support available. For example, the combination of cognitive models and walkthroughs allows assessment of insistence, honesty, predictability and deviation tolerance. This

is because these properties are more user-dependent than the other robustness properties. They can be assessed during specification, but cannot be re-tested until evaluation. This poses a problem in that they cannot be attended to during construction. There is thus a gap between their assessment by analysis and their confirmation by (user) testing. This kind of insight is a further benefit of exposing different forms of interaction between properties and tools/materials.

5.2.3 Internal Properties

The main interactions between tools/materials and internal properties during specification are with development efficiency, modifiability and user interface integratability. Other interactions are minor, such as those with evaluability, portability and maintainability, which are due to positive interactions with architectural models (i.e. appropriate architectures will promote these properties, and thus properties achieved for an architectural model must be preserved during architectural design). Minor interactions here between these properties and architectural models must be preserved during architectural refinement in the software design phase. In particular, modifiability must be carefully considered during all refinements. However, all current model-based tools impose architectures on the final systems, and may thus obstruct properties that interact with architectural models.

Support for external properties can obstruct *run time efficiency* unless steps are taken to counteract this. One possible step is to use *virtual separation* (Shevlin and Neelamkavil, 1991). This can be used to reduce the tension between, for example, observability and run time efficiency by not generating separate coded modules for different levels of abstraction in the final system. Separation is thus *virtual*: it exists during specification but is not preserved in the final system (in Figure 1.4, there would be no separate FCX and UIS as the binding services would create PAC-like agents instead).

Virtual separation lets properties be addressed when they are most relevant during specification. Once established, specification constructs that address them (e.g. link constructs for observability) can be compiled away in the interests of run time efficiency. Returning to more major interactions, *development efficiency* is clearly a key property for specification and design tools. Tools that automatically generate the user interface such as TRIDENT (Vanderdonckt and Bodart, 1993), ADEPT (Johnson *et al.*, 1995), UIDE (Sukaviraya *et al.*, 1993), ITS (Wiecha *et al.*, 1990), and Humanoid (Szekely *et al.*, 1993) improve development efficiency by reducing development decisions. For example, UIDE automatically chooses the appropriate interaction object from interaction specifications.

Tools should deliver known degrees of development efficiency. One way to establish such degrees is to use tools for bench-mark (standard) devel-

opment tasks, and then to assess the speed up over untooled development. For limited tools such as user interface builders, the task for creating a 'hello world' pop-up window is often used to compare their efficiency with toolkits and lower-level libraries.

Benchmarks, however, are only a start. The development efficiency of tools should be assessed for real work (to do this, a software team need a good guess at how long tasks took without tool support). It is questionable to use tools without known levels of development efficiency. For such tool usage to be worthwhile, there must be extensive compensating support for other properties.

Development efficiency is very dependent on the appropriateness of constructs supported by a specification tool. For example, several constructs can be used to specify the dialog level of interaction. However, each construct is biased towards a specific set of dialog requirements (e.g. sequence, interleaving, permutation, ease of walkthrough), and thus a tool based on an inappropriate construct for a system's requirements can obstruct development efficiency. As a simple example, consider simulating interleaved processes with state transition networks. This would quickly bring development to a halt, due to the explosion of interaction points when interleaving two or more processes (see page 139).

A degree of development efficiency can also be *delivered* by reducing the number of design decisions that developers must make. User interface standards attempt to do this by taking many design decisions away from developers. GUI guidelines standardize many features, but mostly for 'look and feel' at the logical interaction level. Example guidelines include CUA (Windows and OS/2), Motif style guidelines (OSF, 1990), and guidelines for constructing Macintosh user interfaces (Apple, 1992). Guidelines may come in printed or on-line versions (Sadler, 1993), and there is anecdotal evidence that the latter format is preferred by developers.

Conversely, development efficiency is reduced when written style descriptions are ambiguous or incomplete. To take a detailed example, the Windows 3.1 Application Design Guide (Microsoft, 1993a) did not specify what should happen on pull-down and pop-up menus when the mouse re-enters the menu with the left button depressed. Developers often copied the common option of ignoring this event and forcing the user to re-select a menu (title). However, this obstructed deviation tolerance by penalizing users who slip off the bottom or side of a long menu. This omission was rectified for the Windows 95 style guide (Microsoft, 1995).

Guidelines are best developed using formal notations, where ambiguities and incompleteness would be easier to detect (Chen, 1993). These could be left as they are for developers who can read them, and re-expressed in natural language for those who cannot. Better still, interaction techniques should be embodied in materials for use at the construction stage (e.g. the

Macintosh tool box, Visual Basic and similar implementations of Microsoft Windows style guides (Microsoft, 1993a, 1995).

In summary, appropriate specification constructs and thorough unambiguous guidelines result in favorable interactions between *development efficiency* and related tools and materials.

The second significant interaction with an internal property during specification concerns *modifiability*. Absolute unconditional modifiability is only encountered in science fiction. Modifiability only extends to known classes of potential change for which a software architecture is known to be suitable.

Further interactions with modifiability result when system designers and software engineers use the same tools to modify a system as they use to initially design and implement the system. The system designer and software engineer (i) modify the original requirements, (ii) modify the specification to reflect the changes in the requirements, and then (iii) modify the code to reflect the changes in the specification.

This three-step process is facilitated if the system designer and software engineer use tools that automatically transform materials at one level into more detailed materials at the next lower level. If system designers and software engineers avoid manually modifying materials at lower levels without modifying the corresponding material at the next higher level, then no system modifications will be lost when the system designer and software engineer use this three-step process to modify a system. Modifiability can thus be further assisted by the use of model-based generation tools. The use of these tools has largely been confined to research teams, although ITS was used successfully to generate the public information system at the Seville EXPO (Wiecha, personal communication).

ITS demonstrates the potential of generators in user interface development. Modeling facilities in ITS proved to be adequate during the EXPO, despite significant changes to the public information system following observation of its usage. Most modifications added capabilities to the functional core (e.g. broadcast of text and images for update to the electronic news service). Since EXPO, three more applications of similar complexity to the EXPO system have been built and developers other than the EXPO team now use ITS within IBM (Wiecha, personal communication).

If tools that automatically transform materials are not available, then the system designer or software engineer must manually change the requirements, the corresponding specifications and the corresponding code. Given that many non-functional requirements are pervasive constraints on design and implementation decisions, it is hard to generate specifications from a full range of requirements. The only tool support for what seems to be an inherently manual activity comes from design rationale tools and from Hypertext links between requirements and design decisions (Kaindl, 1993). Such tools assist modifiability by keeping developers aware of all

the requirements that relate to a design feature under modification. The efficiency of modifications is improved as a result. The need for maintenance will be further reduced by avoiding modifications that adversely affect related requirements because developers were unaware of their relevance when designing the new modification.

User interface integratability can also be assisted by specialized user interface development tools (e.g. UIMS, UI builders). Consider two different applications with different user interfaces that have not been well engineered (for one, there is little documentation, also no tools were used in their construction). If they must be integrated, it may be necessary to reverse engineer design or requirements specifications from working systems. Specification tools assist with the reverse engineering of designs, but this does not provide enough support. For simple applications, and ones with limited interaction, reverse compilers, cross-compilers, and high-level translators are currently used to support integration. However, we know of no such tools that provide extensive assistance for user interface integration. If they did exist, they could perhaps even *deliver* user interface integratability.

Table 5.3 summarizes the interactions between tools/materials and internal properties during specification phases. Internal properties are more evenly covered than external ones during specification, although much of this depends on preservation of properties supported by architectural models.

5.3 Construction Tools and Materials

The coarse phase of construction spans module coding, module tests and integration tests. Properties that have been delivered or proved during specification must be preserved during these phases. Preserving properties from specification requires considerable developer effort if there are no simple equivalences between key constructs in specification notations and the constructs provided by construction tools.

Properties that have been addressed or assessed during specification can only be systematically preserved by formal program transformation. Alternatively, a constructed system must be shown to conform to a specification after each software design decision.

Properties that could not be addressed during specification can be *assisted* at this stage by the use of appropriate materials. The materials here are always some form of re-usable code modules, which can be services within a target environment or capabilities provided by class or module libraries.

Table 5.3 *Specification Interactions between Tools/Materials and Internal Properties*

Property	Interaction	Comment
Development Efficiency	Deliver	UI Management Tools/Builders with validated efficiency, but such tools are rare Model-based UI generators (mostly research and industrial prototypes) Appropriate specification abstractions, but only dialog level abstractions are well established Detailed unambiguous style guides, but these are rare (toolkit implementations for construction are better)!
System Modifiability	Deliver	Architectural refinement, but only for anticipated potential changes Model-based UI generators (mostly research and industrial prototypes) Hypertext requirements linking tools such as RETH (Kaindl, 1993)
User Interface Integratability	Deliver	Limited support from general (UI) tools
Run Time Efficiency	Deliver	Virtual separation
Portability, Evaluability and Maintainability	None	Preservation of property from architectural model

Assisting: a Further Form of Interaction

In Section 5.2.1 the possible interactions between properties and specification T/M were split into five (or six) classes: inspection, delivery, proof, addressing, assessing (and obstruction). It may now be useful to make a further distinction between addressing and a weaker interaction *assistance*, where a T/M gives an implementer *some* constructs that may be used to obtain a certain property.

Assistance is specific to construction, where re-use of code (and tools) is worthy of consideration. As with other forms of positive interaction, it is defined solely in terms of the developer activities required to exploit the interaction. Assistance requires two activities: implementation followed by

specialization. Implementation is an activity specific to the construction phase.

Implementation – developers use extensive knowledge of a property to program an appropriate construct by using general system, programming or algebraic constructs (e.g. implementing Yang's (1988) two-stack undoing model using standard imperative data structures).

Some forms of interaction between properties and tools/materials were introduced in Section 5.2, but assistance is a distinct form of interaction. Assistance interactions are not instances of addressing, because the materials involved require more than basic knowledge to specialize them. Assistance interactions are not instances of neutrality because the materials involved do provide some support. Assistance is thus in-between 'what we want and what we have got'. It is better than nothing, mostly because assisting materials or basic tool constructs can be combined, refined and extended to form appropriate constructs that do address a property.

It is important to distinguish between assistance and addressing. The difference is due to the development activities required for each. To address a property, a tool or material must provide a general construct that only needs to be specialized. This can involve setting attributes, filling in slots, specifying logical conditions or naming functions and procedures to be called to perform a specific function. Specializing a construct generally instantiates one part of a design. Where the construct addresses a property, this instantiation will deliver the property for the corresponding design element.

To assist a property, a tool or material need only provide the means for implementing a general construct that can then be further instantiated to deliver a property. The implemented construct will thus address the property, but it must be implemented. This is the key difference, although it is easy to overlook. With assistance interactions, implementation of constructs that address properties is possible and perhaps even relatively straightforward, but there is no guarantee that its existence or potential will be realized and exploited. Furthermore, constructs that address properties must be implemented and specialized before any assessment is possible. Such assessment will often require users to interact with the system – inspection of coded modules will not suffice.

The difference between addressing and assistance is illustrated by the development of structured programming. Specialized constructs in structured programming languages directly address iteration, selection and procedural abstraction. The provision of stack frames and a run time stack directly addresses the needs of recursion. In contrast, assembly languages only assist with these fundamental control constructs and information structures (selection, iteration, procedures, stack frames for recursion), since jumps, comparisons and stack pointers must be skilfully combined to implement

structured programming constructs. Assembly language programmers still fail to make use of these constructs, with obvious implications for software quality.

Assistance with an external property can clearly vary, but variations will generally be reflected in development efficiency. An overall evaluation of tools and materials can thus distinguish different levels of assistance. With this diffuse form of interaction introduced, interactions between properties and tools/materials during construction can be analysed.

5.3.1 Flexibility Properties

Flexibility properties are rarely proved or delivered before construction. Requirements and specification tools usually only indirectly support these properties by letting developers specify a system's requirements. Construction tools actually implement the materials that establish properties.

Properties that concern the flexible representation of information require extensive run time support. These run time materials must be in place at the outset of construction stages.

Device multiplicity is delivered directly by resource managers, such as that found in graphics libraries and window systems. However, window systems (notably X Window, Scheifler *et al.*, 1992) may restrict devices to a raster display, a keyboard, and a mouse with up to five buttons. Thus while some device multiplicity may be provided, it may not take the form required for a specific system.

When providing device multiplicity, and thus multiple foci of control, various resource managers must cooperate with each other. Support for such cooperation could be handled by a further software component which can be viewed as a resource manager manager. A resource manager manager must handle cross-resource manager transfer and sharing of control, while handling the synchronization of various resource managers. An example application for a resource manager manager is the synchronization of real time video with real time audio, where the cooperation of audio and video resource managers must be controlled by a yet higher level manager.

World Wide Web browsers are now providing extensive support for device multiplicity. Some web browsers adapt layout according to the display device in use. These capabilities have been extended to input widgets in tools such as Sun's Hot JAVA (Gosling and McGilton, 1995).

Representation multiplicity is assisted (for output) by mechanisms such as the View Controllers in SERPENT (Bass *et al.*, 1990). Indeed, these link constructs almost address the property, since representation multiplicity is easily supported by having multiple view controllers for a single functional value. Still, there is no construct to encapsulate the representations for a single (group of) value(s), and thus developers must manage the modularization themselves. In contrast, the DIAMANT UIMS (Trefz and Ziegler,

1989) does have a representation manager that directly addresses representation multiplicity by encapsulating the relevant information.

'Separable' user interface tools support separate specification at different levels of abstraction for interactive systems. They can assist with representation multiplicity for input. The dialog notations in such tools (e.g. UIMSs) may be used to configure alternative user inputs, and also support their translation into a common functional level representation. No further effort from the developer is required. The tool itself preserves properties configured during specification, so developers need not attend to them during construction. However, most current construction support for flexible representation of information is in the form of materials rather than tools.

Moderate assistance for representation multiplicity (output) is provided by the standard Smalltalk methods for Model-View communication. These have been generalized in the paradigm of access-oriented programming and the associated use of *Active Values* (Myers, 1988). When state values are 'active', pre-specified actions are triggered when a value is changed. Active values almost directly address representation multiplicity (and also observability) by letting multiple rendering actions be triggered whenever a value is changed. The actions associated with a value change do encapsulate the multiple representations of that value, but they also encapsulate other behaviors associated with value changes.

Access-oriented programming is largely restricted to research systems. Much more restricted support is provided by typical target environments, for example, application events such as those found in version 7 of the Macintosh operating system (Apple, 1993). Here, different programs can send and receive arbitrary events, once they have registered them, and interests in them have been noted. This is a basic capability that requires detailed development effort, and can reduce development efficiency. However, it could provide direct support for access-oriented programming. For example, in an object-oriented programming language, the assignment operator could be overridden for a class of 'active' objects. The overriding assignment operator would make the assignment, but also call all the methods in a dependency list. These methods may be imperative procedures, or simply broadcast events to notify the value change.

This example demonstrates the difference between different extents of assistance. Neither active values, nor View Controllers, nor application events directly address representation multiplicity. However, each requires increasing levels of developer effort and expertise to create constructs that do address representation multiplicity. If developers are unaware of access-oriented approaches, they may program an event to be raised when the functional state is changed, but they will also have to program the dialog to respond to this event by propagating display changes into logical interaction events. The extent of assistance is thus crucial in any evalua-

tion of tools or materials, since this interaction covers support that almost addresses a property to interactions which come close to neutrality.

Representation multiplicity is given broader, albeit conceptual, support in the PAC model. Specialized PAC components such as Multi-View agents (Nigay, 1994) can support representation multiplicity for output. For input, PAC-Amodeus constructs such as melting-pots (Nigay and Coutaz, 1995) assist in the provision of multi-modal representation multiplicity. However, re-usable materials for these constructs are not widely available, so most developers will have to specialize more general constructs in order to implement these PAC concepts.

I/O re-use is assisted by inter-application communication facilities. Re-usable code for clipboards is one specific example. Such code delivers I/O re-use, as do history modules. An alternative to modules for history support comes from multiple inheritance in object-oriented languages such as Eiffel, where all interactive objects can inherit the capabilities of a history class (Meyer, 1988).

Whatever the mechanism for I/O re-use, there must be compatibility between the source and the target of the re-used information, and this usually must reflect different levels of abstraction. For example, plain text is a material at the logical level of interaction, as its pure ASCII format is device-independent. Re-use of such values is easy to provide, whereas other values present more difficulties. For example, re-use of commands at the functional level is not straightforward, nor is re-use of exotic media, such as real time video at the physical level.

Within single systems, compatibility of re-used information can be addressed, although development effort can be high. However, re-use between systems requires standards (e.g. OLE, Williams, 1994) that allow the re-use of information between disparate systems such as splicing of a video image into a spreadsheet. Extensive re-use is hard without standards. Thus the *LiveText* prototypes developed at AT&T (Fraser and Krishnamurthy, 1990) could achieve only a fair level of I/O re-use on the basis of existing Unix text output conventions. More extensive I/O re-use was seen to require new standards, for example the output of records similar to those found in text editor 'piece-tables'. Such standards would have required major departures from text I/O conventions for Unix commands.

To achieve properties for flexible planning of task execution requires skilled use of materials. For example, groupware toolkits (Gibbs, 1989; Knister and Prakash, 1990; Dewan, 1993) deliver *human-role multiplicity*. Similarly, resource management code addresses multi-threading directly, by letting multiple processes share the same physical devices and related resources, but multi-threading constructs need to be used with skill.

Resource managers also assist in the satisfaction of *non-preemptiveness*, since multiple processes make pre-emption easy to avoid. However, this automatic provision may obstruct the satisfaction of pre-emptiveness when

this is required. This is because non-preemptiveness is always delivered along with multi-threading. Developers must thus implement extra constructs to re-introduce pre-emptiveness.

Reachability can be proved during specification and preserved during construction. Alternatively, reachability can be delivered by construction materials such as re-usable history modules (Berlage and Spenke, 1992). Such modules provide generic capabilities for stepping backwards and forwards through the interaction history, as well as skipping unwanted steps.

Properties that relate to flexible representation of information (representation multiplicity, device multiplicity, I/O re-use) and flexible planning of task execution (reachability, non-preemptiveness, multithreading, human role multiplicity) require extensive run time support. So too do properties that address adaptation of dialog forms (reconfigurability, adaptivity, migratability).

Properties that address adaptation of dialog forms are currently almost entirely supported by materials. Only *reconfigurability* is given extensive support (see below). Other properties are less well supported. For example, *adaptivity* can be provided by specialized artificial intelligence (AI) tools such as User Modeling shells (Kobsa, 1990), but the property requires more extensive support than this. The most successful widespread adaptive approach is 'plug and play' as used by hardware manufacturers to ease the installation of various user interface devices such as mice, video interfaces and audio interfaces. These are intelligent devices that have knowledge of what other kinds of devices can be plugged into the parent system. When the devices are plugged into the system they autonomously determine what other devices are present and using this information configure themselves appropriately so as not to interfere with the other devices. This relieves users from having to manually resolve bus conflicts and similar complex problems. As this affects devices, the property of device multiplicity is also supported, and it provides a much needed alternative to the restrictions of current window managers.

No tools that provide specialized support for *migratability* are known. However, when functions or tasks migrate to the system, some system component must take control. Materials in some form are required. Such components are often called *agents*. These may operate at the dialog level (e.g. Microsoft's Wizards), and thus relieve users of planning decisions. Other agents can operate at the functional level. Software materials could provide re-usable agent 'skeletons', but we are unaware of such support being currently available in any form.

Returning to *reconfigurability*, construction tools and materials support this in many ways, but there is no overall coherence at present. In the past, reconfigurability has been assisted by materials that underpinned the 'table-driven software' approach. Here configuration files hold values that set various system options, such as the right margin setting in the case of a

text editor. This approach requires a module to read configuration files at start up, and to then modify the state at any level of abstraction to reflect expressed user preferences. Each user can have their own configuration file, and thus their own view of how the system should perform.

Reconfigurability can be obstructed by virtual toolkits, especially if they take a lowest common denominator approach. In this approach, the available widgets/controls and their look and feel are restricted to a set of widgets that is common to all the styles for the merged platforms. This minimal set may be so restricted that reconfiguration becomes impossible. Problems here are recognized, with key vendors currently moving away from the lowest common denominator approach. The situation is thus improving, and virtual toolkits should in the future obstruct reconfigurability less than they originally did.

A typical approach to letting users reconfigure systems is provided by the X Window System Resource Manager. It employs a form of table-driven customization (Scheifler *et al.*, 1992). While X-based applications usually use this at system start up, nothing prevents dynamic use of data from the resource database.

Both end-user and developers' tools allow reconfiguration of many features, e.g. window decorations (e.g. scroll bars, command icons, borders), key bindings, mouse button bindings, default fonts and colors, interpretation of various mouse movements (e.g. focus follows mouse, focus changes on click), maximum time between single clicks for them to represent a double click event, and the contents and representation of window manager commands.

It is an important question whether typical users require such reconfigurability. Furthermore, it is not clear that end-users can use all the tools that developers find straightforward. However, the perspective taken in this chapter is one of possibility, rather than ease of learning. Properties associated with ease of learning were not considered in Chapter 2, and thus the learnability of reconfiguration tools cannot be considered systematically in this chapter.

Usability on the other hand can be considered. Reconfiguration tools often obstruct robustness properties such as predictability, since the terms used to describe attributes of window managers (e.g. scroll bars, borders, key bindings, mouse button bindings, default fonts, focus follows mouse entry, focus changes on clicks) have subtly different meanings in different specific commercial products. The result is that the effect of items on control panels and dialog boxes for window manager reconfiguration may be so hard to predict that users give up trying to get any windowing system working the way they want.

Current window systems thus provide considerable support for reconfigurability, but this support is neither coherent, comprehensive nor compre-

hensible. Features have accumulated in a piecemeal manner, with limited thought for the user's view of reconfiguration.

A tool that supports reconfigurability and need not be part of a window system is a macro recorder. This lets users record sequences of actions (keystrokes, mouse movements, screen touches), name, save and edit sequences, and then play sequences back such that they appear to be coming from the user. Such tools relieve users from having to perform complex, error-prone repetitive tasks. However, reconfiguration here is largely restricted to the dialog level of abstraction.

Macro languages and associated script editors also assist in the provision of reconfigurability, but unlike macro recorders, users must program macros themselves. For example Tcl/Tk (Ousterhout, 1994) is a graphical toolkit (Section 5.4.2 below). With it, users can dynamically change many aspects of widgets during system execution. These changes are programmed using a simple command language.

Compared with tools such as Tcl/Tk, users are better supported by UIMSs with customization features, such as S/X Tools (Kühme and Schneider-Hufschmidt, 1992), which provides widgets with several customization options. In contrast, customization options are rarely found in hand-coded widgets for specific projects.

Construction phase interactions with interaction flexibility properties are summarized in Table 5.4. Most support takes the form of assistance for a few detailed approaches to partial delivery of a property. Most exceptions to this are properties that could be proved during specification, which can thus be delivered during construction. However, one property that could not be addressed during specification can be delivered, but only in a limited form (device multiplicity). Overall, support appears to be patchy, with an unprincipled set of local solutions to the challenge of using design principles to guide software development. However, the table does not include properties that can be preserved from specification through the use of model-based UI tools and UIMS. The use of such tools improves support for interaction flexibility during construction, but only for properties that could be addressed during specification (see Table 5.1).

5.3.2 Robustness Properties

Properties for the robustness principle were largely supported by assessment during specification. Few tools or materials assist during construction. Support here is very specific and rarely provides general support for a property. Still, partial support exists for most properties.

Observability is assisted by all declarative constructs (e.g. view controllers) that assist representation multiplicity, with differing extents of support for each tool or material. Observability can be further assisted by context-sensitive help. Such help can tell users what is currently possible

Table 5.4 *Construction Interactions between Tools/Materials and Interaction Flexibility*

Property	Interaction	Comment
Device Multiplicity	Deliver	Resource Manager, but often restricted to specific drivers in window systems, unless Plug and Play supported
Representation Multiplicity	Assist	By View Controllers (SERPENT), but mostly support from materials (e.g. Model-View Controller (Smalltalk), Multi-View Agents)
I/O Re-use	Assist	Inter-Application communication facilities, if compatibility problems avoided Object Linking and Embedding
Human-Role Multiplicity	Assist	By groupware toolkits
Multi-threading	Deliver	Resource Manager
Non-preemptiveness	Assist	Resource Manager, but pre-emptiveness can be obstructed
Reachability	Deliver	By re-usable history module (or class)
Reconfigurability	Obstruct	By virtual toolkits, but situation is improving
	Assist	By table-driven software, macro recording, feature modification (e.g. changing menu items) and tools such as Tcl/Tk
Adaptivity and Migratability	Assist	Limited support from materials (e.g. User Modeling Shells, Plug and Play, AgentWare?)

and how to accomplish it, thus making the current state of the user interface observable. When a broad range of context-sensitive help facilities is encapsulated in a re-usable module, then this is a specific form of material that addresses a small part of observability.

Insistence is delivered in one specific form by materials that implement modal dialog boxes in toolkits, and in other materials that implement repeated replay of audio until some user acknowledgement.

Honesty is assisted by all declarative constructs that assist representation multiplicity, and by all materials that improve response time, as users can perceive a system as lying when known 'out-of-date' information stays rendered. It also requires good response times even at the physical level of interaction, where immediate character-by-character feedback during typing is preferred to delayed output. Many aspects of honesty however are given no support, for example the suppression or revision of warnings and error messages that no longer hold (because the monitored condition has changed).

Access control is delivered in a broad form by access control lists that hold information on access to data and commands by user roles. There are materials that provide some basic assistance with access controlability. For example, (the code for) a file system manages read, write, and/or execute permissions. More extensive support can be envisaged, and is provided in part by Suite (Dewan and Shen, 1992). Instead of providing access control in the back-end or persistent store of an application, Suite implements access control in the front-end or user-interface of the application. As a result, it is able to provide earlier feedback to access violations and protect fine-grained operations (such as move cursor) on logical user-interface objects (such as paragraphs) instead of coarse-grained operations on physical objects (such as files). Such support is important in collaborative environments.

An alternative (or complementary) approach is to have appropriate modules form a framework for integrating single-user legacy applications into a multi-user cooperative environment. The framework could route information between applications without any 'knowing' it is being used in a new multi-user environment. The COLA approach developed at AT&T has created a systems programming basis for such a framework. Extensions to standard Unix library functions such as open, read, write and close let these be treated like active values, with other actions being triggered whenever they are called (Krell and Krishnamurthy, 1992).

Predictability is delivered in one specific form by percent-done indicators (Myers, 1985) (and the system is more honest as a result). Response-time stability aspects of predictability can be supported by materials that let developers reduce resource usage (paging managers, hypertext pre-fetch code, dynamic linking and indexing code). Such materials can also improve the *pace tolerance* of the system, by reducing adverse system delays.

Pace tolerance is not just concerned with shortest possible response time of a system. Users also need to control the interaction pace, such as specific capabilities for controlling mouse acceleration and setting double-click intervals, as well as operations for designers to insert delays. Such specific support currently comes in the form of materials (i.e. library routines) with limited tools (control panels, resource editors). UIMS generally lack time constructs. For example, RAPID only had a time-out construct (Wasser-

man, 1985). Pace tolerance is also delivered by operations for introducing delays into the interaction. Materials that calculate the time needed to read a message before removing it automatically using Bevan's algorithm (Bevan, 1983) would also make a small contribution to pace tolerance.

However, little attention has been paid to *deviation tolerance* when reading set-up files. The simple database manager can detect misnamed parameters and inappropriate value settings, but there is no provision for error recovery by the system or the user (there are not even notifications of problems).

Some programming languages have fail-safe features (e.g. error handlers in Visual Basic; Microsoft, 1993b). Such features assist deviation tolerance, by providing an infrastructure for implementing error detection and recovery. Further support is provided by materials that implement error recovery from either user or system errors. In the case of system errors, re-usable checkpointing and roll-back code can be used. Many database tools can provide these capabilities at the functional level of an interactive system (e.g. the database capabilities of Visual Basic). The most robust tools maintain their checkpointing logs and repositories independently of the system to ensure they will not be contaminated by system failure. After a system failure, the system state can be set to that of a selected backup, or a selected log can be processed to reach the desired system state.

In the case of user errors, re-usable modules that implement undoing capabilities can be used (Yang, 1988). Some implementation frameworks assist with the provision of this feature by providing basic support for undoing. For example, Command objects in the MacApp framework (Schmucker, 1986) can have undo methods associated with them. However, the developer must construct an inverse for each command to take advantage of this. Even so, this is still assistance with the property of deviation tolerance, despite its very basic and specialized nature.

Suite provides automatic support for (multi-user) undoing of user manipulations of (distributed) active values (Dewan and Choudhary, 1995). Any side-effects taken in response to these modifications by the application must be undone by application-defined undo methods. Thus, the responsibility for undo is divided between the generator and the application with the generator undoing its actions and the application undoing the ones it takes.

Construction phase interactions with robustness properties are summarized in Table 5.5. Compared to flexibility, more support takes the form of delivery, but this is again in the form of local specialized solutions that partially deliver a property. Several properties could be assessed during specification, but little can be done to preserve this during construction, other than by formal transformation methods and for the limited solutions provided by specific materials. However, the table does not include properties that can be preserved from specification through the use of model-based UI tools and UIMS. The use of such tools improves support for interac-

Table 5.5 *Construction Interactions between Tools/Materials and Interaction Robustness*

Property	Interaction	Comment
Observability	Assist	Generally, T/Ms supporting representation multiplicity (e.g. view controllers) support observability Also assisted by context-sensitive help and UIMS with Arch/Slinky architecture
Insistence	Deliver	By very specialized materials (e.g. materials for modal dialog boxes or repeated audio replay)
Honesty	Assist	Generally, T/Ms supporting representation multiplicity, response-time stability and pace tolerance support honesty
Predictability	Deliver	Percent-done code delivers partial and very specialized support (response-time conformance, also achievable by reducing resource usage)
Access Control	Deliver	By access control lists
	Assist	By customized overlays as well as by more basic file system features
Pace Tolerance	Deliver	Delay introducing operations (e.g. for reading messages)
Deviation Tolerance	Assist	By 'clean' dialog abstractions that support processes, by constructs for error recovery such as fail-safe programming language features
	Obstruct	By resource managers that silently ignore errors in configuration files (e.g. X Window System)

tion robustness during construction, but only for properties that could be addressed during specification (see Table 5.2). The table also omits interactions with logging code, as that is largely outside the scope of this chapter. However, the logs produced will highlight adverse patterns of user interaction, especially failures in robustness.

5.3.3 Internal Properties

Much support for internal properties comes from the architectural model, but such properties must be preserved in the final architectural refinements and throughout construction.

Construction tools should always improve *development efficiency*. This can be compromised if the underlying configuration or programming language is not well formed and properly specified. For example, a problem with the concrete syntax for subnet traversal in RAPID forced cumbersome 'fixes' when a subnetwork needed to be traversed from more than one point in a calling network (Cockton, 1985).

Instrumentation code profiles the space and time consumption of executing processes. It delivers some *maintainability* (by highlighting adverse resource usage) and some *evaluability*. Its main value is in its assistance for maintainability, where it helps to discover errors (from failures to meet requirements to system crashes), and to locate the cause of the error. Instrumentation code however does not assist with the key step in maintenance, i.e. correcting the cause of the error. Tools for discovering errors include:

- video and audio recording facilities that capture user activities and commentary;
- quality assurance testing procedures which validate that the system meets its requirements;
- debugging tools for conducting tests and experiments to locate errors;
- performance and resource monitoring tools;
- logging and evaluation tools (tools for evaluability).

Tools for locating the cause of the error may include profilers and testers. Tools for correcting the cause of the error include text editors and specification/programming tools to correct the error and regenerate the appropriate materials.

At the same, instrumentation code does obstruct *run time efficiency*. Thus properties need to be traded-off when selecting tools and materials, just as they had to be when selecting architectural models.

Further support for assessment of internal properties such as maintainability and modifiability comes from the use of inspection techniques. Remaining properties are only assisted. For example, *evaluability* is assisted by materials such as code that logs invocation of each event. Tools that analyse such logs belong to the evaluation phase.

Portability is supported by the use of code supporting layers and wrappers around platform dependent features, or the use of emulators and simulators which let systems coded for one specific hardware and/or software environment execute in another one (this approach may obstruct run time efficiency, but it can 'buy time' for a more thorough conversion of the

application). For new applications, virtual toolkits (Retter *et al.*, 1992) deliver portability.

Run time efficiency is supported by *virtual separation* (Shevlin and Nee-lamkavil, 1991), where any execution inefficiencies due to separating levels of abstraction at design time can be removed by tight physical integration of the run time code for the user interface and the functional core. Virtual separation in this sense is little more than the specialization of compiler optimizing techniques for separable interactive systems development.

User interface integratability has two distinct aspects. On the one hand, the user interfaces of separate architectural components should be consistent and interoperable. On the other hand, it should be possible to compose separate interactive components into a single system.

Consistency and interoperability for interactive components are supported by materials that implement components described in style guides (Microsoft, 1993a, 1995). User interfaces that use common components will be easier to integrate. Some components cover all levels of abstraction in interaction. For example, Visual Basic's common dialog boxes (see Section 5.4.3) are common functions that have been factored out of the individual logical, dialog and functional levels of their interactive behavior. Visual Basic provides such common dialog boxes along with some underlying functionality. Other construction tools provide support for composing dialog boxes and related functionality. Most X toolkits (e.g. Tcl/Tk; Ousterhout, 1994) supports 'superwidgets' that are such compositions.

'Composability' of interactive components requires basic software support in order to address this aspect of user interface integratability. Basic assistance is provided by materials that implement inter-application communication protocols. Users can use these to share data among these applications. For example, users may cut, copy and paste data between applications. Many object-oriented computing environments now support embedded objects, which are constructs that assist user interface integration. For example, Microsoft's OLE (Williams, 1994) lets users of one application invoke functions provided by another.

Another form of support is found in Field (Reiss, 1990), which addresses the composability aspect of user interface integratability. In Field, every user interface broadcasts events of interest and other user interfaces can register interests in them. For instance, a debugger can broadcast the statement being executed and an editor can receive it and then highlight the current line. This has supported integration in programming environments (by putting minimal wrappers around tools). The method is now in commercial use.

Lastly, one may also regard X pseudo servers (Lauwers and Lantz, 1990) as delivering user interface integratability by integrating multiple instances of the same interface without requiring any changes to the user interface.

Construction phase interactions with internal properties are summar-

Table 5.6 *Construction Interactions between Tools/Materials and Internal Properties*

Property	Interaction	Comment
Development Efficiency	Deliver	Well-designed tools and materials should always deliver this property
System Modifiability	Assess	By inspection techniques, but largely an architectural property
Portability	Assist	By virtual toolkits and more generally by layered wrappers or emulations and simulators
Evaluability	Deliver	Instrumentation code
Maintainability	Deliver	Instrumentation code reveals common problems
	Assist	By Inspection Techniques
Run Time Efficiency	Obstruct	By Instrumentation code, layered wrappers and emulations/simulators, which slow things down
	Assist	By virtual separation, which removes layers at run time
User Interface Integratability	Assist	By standardized (style-guide-based) components and other common components By tools such as Visual Basic (Microsoft) and Tcl/Tk By materials such as inter-application communication facilities and Object Linking and Embedding (Microsoft)

ized in Table 5.6. There are no obvious patterns in the table, other than a wide range of forms of interaction. However, the interactions noted here are clearly only a sample of possible ones, since there are many general software tools and module/class libraries that offer favorable interactions with internal properties. The analysis above has thus highlighted the more novel interactions that are especially relevant when constructing interactive systems.

5.4 Commercial Tools

Three commercial tools are now analysed in depth to further validate and extend the analysis from the previous two sections. They are 'commercial' in the sense that they are either products or have a widespread user base.

There are two main uses for a tool study. On the one hand, a full analysis of properties would be a (complete) tool evaluation, but on the other hand a more restricted analysis can confirm existing and expose further interactions between properties. The restricted analysis can also expose the complex ways in which a tool may interact with properties. All examples in the analyses below are chosen with the second use of a tool study in mind. They are not to be taken as complete and balanced evaluations of each tool. Instead, they reflect the interest of the authors in the utility of the properties and architectural analyses developed in earlier chapters. They are thus more evaluations of the value of property profiles and architectural analysis than summative evaluations of the worth of the three example tools, which are TAE+ (Szczur and Sheppard, 1993), Tcl/Tk (Ousterhout, 1994) and Visual Basic Version 3.0 (Microsoft, 1993b).

External properties are visible to the user of a system. This means that users of the interactive system being developed will be aware of them. Likewise, external properties of the tools used during design and development are visible to developers (as tool users). Tools therefore manifest external properties to developers and more or less support internal properties being designed into the system being developed. Each of the following examples looks at these differing aspects of interaction between tools and properties.

5.4.1 TAE Plus

NASA's Goddard Space Flight Center develops and maintains software to provide for control of all NASA's unmanned spacecraft and for the collection and analysis of the resulting scientific data. The Transportable Applications Environment Plus (TAE+) was designed to handle the development of user interfaces and the run time management of systems in this complex, heterogeneous, distributed computing environment. TAE+ is now distributed commercially by Century Computing.

TAE+ supports the Motif (OSF, 1990) look and feel for a wide variety of platforms. Developers use the TAE+ Workbench to specify the layout and dialog of a user interface. The application's windows are constructed from Motif widgets and presentation types that are combinations of Motif widgets. The Workbench generates a resource file and code to implement the user interface in ANSI C, K&R C, C++ or Ada. Developers add functional core routines to complete the application.

A set of application services, the Window Programming Tools (WPTs), provides run time support, managing the user-interface layout and the

dialog that has been specified in the Workbench. In addition, the user interface may be dynamically updated by the application using a run time interface library. Communication between the Workbench and the run time support system occurs via the resource file.

The discussion below evaluates TAE+ very briefly in terms of the internal properties of Chapter 2 and concentrates in more detail on the external properties of Chapter 3. For the most part the evaluation is done in terms of the TAE+ user (an interface designer) as opposed to the end user of an application developed with TAE+.

Modifiability

TAE+ partitions an application into three distinct parts: layout, link-based dialog, and functional core. Each part may be modified separately. For instance, if only layout changes are made, the application may be restarted with the updated resource file. The system does not need to be rebuilt.

Similarly, program code may be attached to the link-based dialog within the Workbench. This code may be modified externally and the changes will be maintained when the system is regenerated from the Workbench.

Portability

TAE+ supports the following platforms: Sun (SunOS and Solaris), Hewlett-Packard 9000 series (HPUX), Silicon Graphics (IRIX), IBM RS/6000 (AIX), Concurrent RT-7000 (RTU), Intel 486-based (SCO Unix and Linux), DEC station (ULTRIX and OSF/1), and DEC VAX (VMS). A user interface generated for one platform is completely transportable to any other.

Evaluability

Two tools help TAE+ usability engineers evaluate the 'goodness' of TAE-produced end-user application interfaces. An adjunct tool, CHIMES, can be used by the user-interface designer to check consistency across windows (e.g. placement of objects) and compliance of layout with usability guidelines (such as number of colors and type fonts). A second (adjunct) tool, the User Action Graphing Effort, uses TAE's Perl scripting capability to capture data used to compare the actions (keystrokes, mouse clicks) taken by a novice in performing a task to those of an expert doing the same task. A graphical display of the actions of the two users and a time-stamping capability enable a usability engineer to identify features of the user interface that need to be made easier to use. TAE+ support for rapid prototyping further improves evaluability when combined with co-operative evaluation (Monk *et al.*, 1993).

Maintainability

The scripting facility described above can be used to develop test scripts and an automatic application test suite. TAE+ thus supports maintainability (and modifiability) by addressing *regression testing* (the repetition of tests passed by code before it was changed). Successful maintenance requires that changed code should pass these tests again.

Should user problems indicate a need to change displays, then layout changes can be made quickly and take effect without rebuilding the system.

Dialog changes that affect only the resource file may also be accomplished without rebuilding the system. TAE+ lets developers make run time changes in the user interface through library calls from the functional core. While this feature extends the range of interfaces that can be developed, it negates the separation of the functional core from the user interface. This may result in increased maintenance costs.

Run time Efficiency

The run time efficiency of TAE+ is dependent on the operating environment. Applications running locally on up-to-date workstations seem 'fast enough'. As with all systems, network delays or out-of-date hardware can cause problems.

User Interface Integratability

TAE+ generates applications with the Motif 'look'. Motif operation guidelines are enforced to the extent that standard widgets (e.g. radio buttons) are used, but other guidelines, such as menu structure, are not enforced. TAE+ provides the flexibility to model a user interface on existing ones, but this flexibility leaves the designer with the responsibility for compliance on style issues. Local standards can be supported because designers can modify the Workbench, changing the set of widgets that are available and customizing property defaults (color, widgets, etc.).

Functional Completeness

TAE+ can be used to create interfaces that look and feel like those created with Motif-based tools. Additionally, TAE+ implements a number of widgets designed for use in control panels. These include: dynamic text whose color and text string are dependent on threshold values of an attribute, a strip chart, a rotator for circular gauges, a discrete widget that displays unique pictures for a finite number of attribute values, and a mover that animates a defined area of a picture in response to changes in attribute values. These widgets can be activated by user inputs as well as internally-generated data.

These somewhat esoteric widgets are essential for application programs

at NASA Goddard Flight Centre. Basic toolkits such as Motif do not provide this functionality, so these TAE+ extensions make it possible to achieve functional completeness.

For the developer, there are some remaining inelegancies that obstruct functional completeness (e.g. support at run time, but not in the Workbench for geometry management and sub-panels of Dialog Boxes etc., see below).

There are several issues associated with the current release, Version 5.3, of TAE+ in terms of functional completeness. Firstly, the widgets are not all simple Motif widgets; some are a combination of Motif widgets and the Dynamic Data Objects that are unique to TAE+. Although the developer can code at the Motif level, it is currently not simple to do so. It is difficult to add widgets because such additions require modifications to both the Workbench and the API.

Secondly, TAE+ does not support geometry management. Therefore, widgets such as the Motif RowColumn widget can only be used by declaring an X Window workspace in TAE+. (This workspace is not managed by TAE+, but by making windowing system function calls.) Third, the Workbench does not allow a designer to create a panel contained inside of another sub-panel. The run time library supports subpanels – there are plans to eliminate these shortcomings in the next release.

Development Efficiency

The TAE API has been shown to be efficient in terms of learning and coding time because the Window Programming Tools operate at a high-level of abstraction. Further development efficiency results from support in the TAE+ Workbench for object re-use through copy and modify.

TAE+ provides a Rehearse function that permits the designer to prototype the user interface and provide clients with an operational prototype without coding. The prototype may be used for design reviews and successive refinement of the interface before commitment to a final design. This has been shown to reduce overall development time. There are several features that contribute here, for example, being able to make layout changes without rebuilding the system. However, if user needs must be addressed by adding new widgets, then development becomes less efficient (see user interface integratability, above).

Flexibility and Robustness

TAE+ supports many internal properties. External properties are also supported for developers (in the Workbench) and for the user (in generated systems). Thus, end user applications can be developed with TAE+ to meet many of the criteria related to flexibility and robustness, although there is

no explicit CSCW support for the properties of *human role multiplicity* and *access control*.

Two flexibility properties associated with planning of user actions (i.e. *reachability*, *multi-threading*) and robustness properties associated with the current state of the system (i.e. *observability*, *insistence* and *honesty*) can be addressed for those aspects of the dialog that are configured as TAE+ *link-based dialogs*.

Links (connections in TAE documents) are a type of event-response rule. The events are limited to things like selections and field completions. Responses can be to alter a panel (window, dialog box) attribute such as visibility and/or to initiate a call-back. They let designers define simple dialogs without writing code, such as popping up windows/dialog boxes or closing them.

Theoretically the link-based dialog can be analysed for *reachability* and *observability*. As long as the designer uses the link-based dialog, it is possible to check that there is a path which will display all relevant data. However, each event that can have a link associated with it can also generate a call-back to the functional core, or perform (hidden) dialog actions. As most applications require some use of the call-back mechanism to complete their dialog definition, this also means that automatic analysis would be incomplete without analysing the code – a nearly impossible task. Assessment of *reachability* and *observability* in the final system is thus only supported for the exclusive use of link-based dialogs and data-driven objects. The latter let designers easily build an object that changes state when the value of a monitored variable changes. For example, a numeric output can be displayed with three colors: red for out-of-range error, yellow for near out-of-range, and black for in-range. *Insistence* is thus addressed by this construct.

TAE+ supports interruptible behavior in normal operation. *Pre-emption* by the functional core is possible (e.g. error conditions). However, functional core initiated states are not represented in the link-based dialog. Thus, while updates initiated asynchronously by the functional core address the temporal requirements for *honesty*, they further obstruct the analysis of *reachability*. The mix of positive and negative interactions is a good example of the complexity that can arise in property-oriented analyses of tools.

TAE+ currently supports keyboard and mouse input and has been instrumented additionally for speech recognition and synthesis, and thus provides moderate *device multiplicity*. *Representation multiplicity* is achieved with a robust library of objects for representing information, both textual and graphical, and additional objects can be created. For *I/O re-use*, the Workbench supports cut/copy/paste of all objects (automatically renaming them). Only the X Window standard cut/copy/paste are supported in applications.

Reconfigurability can be supported (e.g. in the form of feature modifications) by calls from the functional core to the run time user interface. As the Workbench itself is a TAE+ application it can be readily reconfigured by developers.

The general experience with favorable run time efficiency extends to *pace tolerance*.

Deviation tolerance is given basic support for 1-level undoing.* TAE+ supports restricting the range of numeric fields, but generates an out-of-range error at run time, which could obstruct deviation tolerance.

5.4.2 Tcl/Tk

The interface building tool Tcl/Tk consists of a programming language, Tcl (Tool command language) together with its associated X Window toolkit, Tk. Being a full programming language, Tcl itself is inherently neutral with respect to the external properties proposed in Chapter 2. However, using Tcl together with the interface building facilities provided by Tk, it is possible to build complete applications which provide any desired combination of properties. Alternatively, Tcl/Tk can be used to develop tools with which a user may build systems. With this approach system-building tools can be produced which guarantee a particular set of properties for any resultant system. While any of the external properties could be delivered in such systems, the features and facilities provided by Tcl/Tk, particularly those provided to manipulate the Tk widgets, affect the ease with which the developer might achieve certain properties. Overall, tools such as Tcl/Tk do not address as many properties as systematically as do sophisticated tools such as TAE+. The main consequence is a loss of development efficiency for the more demanding aspects of user interface design.

As with other X Window toolkits, *device multiplicity* in Tcl/Tk is restricted to the use of display, mouse and keyboard. *Representation multiplicity* is facilitated by the provision, within Tk, of a variety of basic widgets. As both Tcl and Tk are designed to be extensible, these widgets can be combined or extended, and more complex widgets created. Thus tools or applications can be produced which deliver the required level of representation multiplicity. The selection retrieval mechanism associated with Tk widgets simplifies the delivery of basic *I/O re-use* such as 'cut', 'copy' and 'paste'. In addition any input/output re-use at the physical or functional levels can be delivered with some programming effort. Although Tk provides no explicit support for *human role multiplicity*, several Tcl/Tk applications have been built which deliver this property, including database systems which support the differing roles of Data Manager, Data Provider

* In 1-level undoing, only the last change can be undone, so an undo followed by an undo undoes the undo.

and Data User, and computer assisted learning systems which distinguish between the roles of teacher and student (Newman and Smith, 1995).

Reconfigurability is facilitated for the developer by the ability to set or alter key bindings, mouse button bindings, mouse movement interpretation and defaults for font and colors that can be set for each object class in Tcl (the latter capability also addresses the internal properties of *maintainability* and *modifiability*). Applications or tools can then be built which allow the user to customize any of these features. Tcl/Tk is neutral with respect to the properties of *reachability* and *non-preemptiveness*, while *adaptivity* and *migratability* could only be delivered with considerable programming effort.

Robustness properties are largely design and specification issues, hence construction tools interact with these properties less than do specification tools. Nevertheless, Tcl/Tk provides some features which may support the delivery of some of these properties. Firstly, the availability of modal dialog boxes, flashing icons and window 'grabs' can assist in the delivery of *insistence*. Secondly, the ability to disable and 'grey out' buttons or menu items, and the ability to change the cursor according to the user's context can contribute to *honesty* and *predictability*.

5.4.3 Visual Basic Version 3.0

Visual Basic is the name given by Microsoft Corporation to its development environment for a version of the Basic programming language that exploits capabilities of their Windows operating systems.

Visual Basic determines the appearance and behavior of a user interface in two ways:

- Some features are determined by specifying them interactively.
- Other features are set during the execution of the Basic program.

Visual Basic's documentation calls these *design-time* and *run time* settings respectively. The values that can be set at design-time and run time are not identical, but there is considerable overlap.

Visual Basic combines features of construction and execution tools. The development environment supports design-time creation of *forms*, which can be used as dialog boxes, document windows or application windows.

Controls (the widgets of user interface toolkits) can be placed on forms. There are controls for text entry, value entry (sliders and spin-boxes), value selection (list and check boxes, option buttons) and command initiation (command buttons, drop-down and pop-up menus). Controls have attributes that affect their appearance and behavior.

Attribute values can be set at design-time. Text labels and icons are treated like controls, which lets their attributes be set at design-time.

At run time, controls respond to a fixed set of input and system events.

Handlers for these events are programmed in Basic. Other Basic procedures can be written, and these can be called from event handlers. These may interrogate and alter the values of attributes as required.

There are many capabilities in both the design and run time environments. The analysis of properties which is given here merely addresses some of the most important of them in relation to the architectural model described.

Examples have been chosen to illustrate three uses for tool studies: tool evaluation; to confirm existing and expose further interactions; to expose the complex ways in which a tool may interact with properties. The analysis below is based on reports from a few of the authors about their experiences in using Visual Basic version 3.0, supplemented by an extensive study of the generally candid programmer's guide (Microsoft, 1993b). All page references below of the form (VBPG xxx) refer to this guide.

Flexibility Properties

There is a clear pattern in the support offered by Visual Basic for flexibility properties, since its run time architecture largely addresses the logical interaction component. It thus lacks most of the functional partitions adopted for architectural analysis introduced in Chapter 4. Since properties that concern flexible planning of interaction depend heavily on dialog functions, the absence of a dialog component affects support. Similarly, most properties that concern flexible representation of information rely on several functional partitions. The lack of clear dialog and functional core adapter components means that such properties cannot be systematically addressed (since they involve interactions via the dialog between the logical interaction and the functional core adapter).

Support for flexible interaction planning is thus largely restricted to logical interaction features. The underlying event model makes it very easy to write modeless interfaces, and thus *multi-threading* and *non-preemptiveness* are assisted. However, non-preemptiveness is easily obstructed by poorly written applications.* To achieve full *user-oriented* non-preemptiveness for all applications running in a Windows environment, every application has to regularly surrender control via a *DoEvents* call (VBPG 417). This reveals the lack of process or equivalent constructs. In terms of the interactions between tools and properties introduced earlier in this chapter, multi-threading is assisted rather than addressed, since process constructs have to be built on top of basic events. There is thus a strong risk that extensive multi-threading will not be achieved when developing with Visual Basic.

* The user-oriented use of *non-preemptiveness* is potentially confusing here, as it is used from the user's point of view, whereas in operating systems it is the currently active process that is not preempted in a non-preemptive environment.

The control constructs of Visual Basic are restricted to event handlers and procedure calls. This restriction results in a monolithic run time architecture, with no modularisation of processes or threads. The analysis of *reachability* is effectively obstructed by the lack of a central dialog abstraction to analyse. Furthermore, the lack of a functional core adapter rules out coarse-grained reachability analysis. It is thus almost inevitable that proofs of reachability will not be attempted when developing with Visual Basic, unless separate dialog specifications are either prepared in advance or reverse-engineered from the code. The latter approach is difficult and error prone.

The lack of a well structured architecture restricts support for representation multiplicity to piecemeal provision of several specific capabilities. Thus, for example, there are several date formats (VBPG 162) and the icon displayed during dragging can be changed (VBPG 279). More generally, semantic feedback during dragging is greatly assisted by the provision of enter and leave events (as discussed in Chapter 4, page 113). Even so, there is no generalized support for simple user interface animation, which is often obstructed by the very primitive event timing in Visual Basic's kernel. An increasingly common form of user interface representation is thus not well supported.

Support for representation multiplicity covers a broad but uneven spectrum. The lack of a complete software architecture for interactive systems forces compensation to take place as extension to the facilities of Visual Basic – but outside it (VBX files: VBPG 123). VBX files provide a way to add new controls (e.g. graphical command buttons with a 3D look and feel). This (rather indirect) assistance for representation multiplicity has led to a proliferation of third party controls. However, development of new controls within Visual Basic itself is difficult, as there are many graphics primitives and attributes that can only be created/set at run time (e.g. graphics methods for arcs and setting pixels (VBPG 339)).

Representation multiplicity is thus addressed for a few presentation features, but is at best assisted and may be obstructed. VBX files let missing features be added, but they do not let unsuitable ones be fixed. Representation multiplicity is restricted in the Multiple Document Interface, as Document Windows (but not dialog boxes) must go inside the parent application window (VBPG 297). Extensions that overcame this restriction would have to re-implement a major part of Visual Basic itself. Interestingly, the Windows 95 user interface has preserved the Multiple Document Interface – reluctantly, as it appears from the style guide (Microsoft, 1995). This strongly suggests that re-programming the Multiple Document Interface requires resources beyond those available to most application and tool developers.

Support for *reconfigurability* is largely similar to that for representation multiplicity. Features such as multinational data formats address both

properties, but assistance, neutrality or obstruction are more common interactions between Visual Basic and flexibility properties. Very low-level language features assist with reconfiguration (e.g. arrays of controls let controls be added and removed at run time). However, some features can only be set at design time. This obstructs reconfigurability by ruling out run time changes. For example, the multi-line text property and scroll bar properties can only be set at design time (VBPG 40). Also elements of control arrays that were created at design time cannot be removed at run time (VBPG 71). Such *ad hoc* boundaries between design and run time have a negative impact on other properties (see below).

Support for other flexibility properties is limited. *Device multiplicity* is obstructed by the absence of multi-media support. *Human role multiplicity* is not addressed in any way (interactions are thus neutral). However, there is some useful basic assistance for *migratability*, as keystrokes can be passed on to other applications, so agents could be implemented that take responsibility for some tasks. As the receiving application cannot distinguish between user- and application-generated keystrokes (VBPG 521), a requirement for migratability identified in Chapter 3 (page 83) appears to be satisfied, but this requirement is user- rather than system-oriented. In fact, commands that have been migrated do not require presentation (i.e. activating main application window, popping up dialog boxes). However, this will happen when migration is driven at the logical interaction level. Properties such as *pace tolerance* and *honesty* will clearly be obstructed by this approach to migration.

Lastly, *I/O re-use* is given basic assistance by clipboard capabilities (VBPG 405) and the ability to pass on keystrokes to other applications (VBPG 521).

Robustness Properties

As with flexibility properties, robustness properties that depend on several architectural components are not well supported. The lack of clear dialog and functional core adapter components means that *observability*, *insistence* and *honesty* cannot be systematically addressed (since each relates to interactions via the dialog between the logical interaction and the functional core adapter). The result is that interactions with these properties are generally neutral.

The remaining robustness properties of predictability, access control, pace tolerance and deviation tolerance are less dependent on extensive architectural support. Even so, Visual Basic provides limited support.

Access control fares relatively well. Database features address it with the capability to restrict read or write access to data items (VBPG 461). However, file operations provide no such support.

Pace tolerance is generally obstructed. At the logical interaction level,

mouse move events may not be generated for each pixel (VBPG 269). This will cause problems for some fine sketching, drawing, dragging and region selection tasks, since Visual Basic may not be able to keep up with the user. A more general problem arises when Microsoft's DLL (Dynamic Link Library) mechanism is used for integration, because the default time-out for data link accesses is five seconds (VBPG 503)! This suggests that such delays are to be expected. DLL access to functional core values during closed-loop interactions, such as slider manipulation during star field queries (Ahlberg and Schneiderman, 1994) will thus result in pace tolerance problems.

Deviation tolerance is given better support, since the database rollback methods provide some assistance with error recovery (VBPG 478). Similar assistance with error detection is provided by the Data Error event (VBPG 475). However, this focused support for error handling is not matched by non-database features. Visual Basic has an 'on error' construct (VBPG 238), but the assistance provided by this and related constructs – resume construct, null return values (VBPG 164) – are too general to provide effective support for deviation tolerance.

Internal Properties

All things being equal, the design-time capabilities result in *high development efficiency*, especially for systems where the functional core is little more than a database. For example, there is a unified SQL interface for all database systems which are supported (VBPG 483). Other capabilities greatly accelerate the development of a few specific functions. For example, the grid control manages rows and columns for spreadsheet and other tabular presentations. The text box, check box, label, image and picture box controls can all be *bound* to database items (VBPG 462), automating the implementation of dialog links between values in the functional core and the logical interaction. There is also extensive support for later life-cycle activities such as installation (VBPG 573).

When these focused features such as SQL interfaces, grid controls and active data values are inadequate, development efficiency is reduced whenever a key external property is inadequately supported. This problem may be alleviated if there is compensation from third party shareware or commercial custom controls. Thus, development efficiency is reduced when 'graphics with semantic content' (in window graphics) are called for, as these must be written from scratch – jeopardizing *functional completeness* unless appropriate custom controls can be purchased. Where complex dialogs are required, this can easily result in large amounts of spaghetti code, reducing development efficiency and *maintainability*, as well as risking functional completeness due to errors on dialog logic.

Development efficiency is further reduced when run time capabilities are

inaccessible at design time. For example, graphics methods (the procedures called to produce graphics) have more extensive capabilities than design-time graphical controls. Similarly, rapid prototyping is obstructed by the inability to place text in grid cells at design-time, since mock-ups of possible tabular displays must be programmed rather than specified interactively.

Lastly, some language abstractions are too low-level to allow rapid development: items must be added to lists one at a time (VBPG 52), bit fields are used to represent mouse and keyboard status (VBPG 271), and explicit indices are needed to set the 'tab order' for the controls for a form (VBPG 66 – third party tools do support more direct specification of this order at design time).

Maintainability and *modifiability* are addressed by a range of general software techniques: modules (VBPG 126), objects (VBPG 181), generic objects (VBPG 132), and public and private procedures (VBPG 132). Specific Visual Basic features also address modifiability. VBX files have already been mentioned, as have control arrays, which ease modification of the set of controls on dynamic forms. However, some arbitrary restrictions limit the effectiveness of some of the constructs: objects cannot be placed in huge arrays (VBPG 175) or in user-defined types (VBPG 183). More generally, maintainability and modifiability are also hindered because the code is spread out in many procedures for many objects, and it is difficult to have an updated overview of the code in a development.

Features that directly address *run time efficiency* place minimal demands on programmers. For example, bitmaps can be compressed using run-length encoding, which saves storage (VBPG 262) and image box controls allow efficient display of images that do not require the full functionality of picture controls. However, the advice in the chapter on run time efficiency (VBPG Chapter 11) is somewhat piecemeal and does place considerable demands on programmers' memories. Such 'tips and tricks' approaches to run time efficiency must have a negative impact on development efficiency. Furthermore, there are some inefficiencies for which no work-arounds are suggested. For example, it can take 'several seconds' to create an OLE object (VBPG 529), which will be unacceptable in many interactive applications.

User interface integratability is well addressed in Visual Basic, since the Windows environment has directly addressed this property in its provision of DLLs (VBPG 493) and OLE. However, Visual Basic places some limitations on OLE parameters that could limit either user interface integratability or development efficiency (VBPG 554).

Support for *I/O re-use* is also relevant to user interface integratability (clipboard: VBPG 405; sending keystrokes: VBPG 521), and standardization of Windows features supports user interface integratability. Visual Basic provides implementations of common dialogs (open, save as, print, color, font) in the CMDIALOG VBX file (VBPG 103 and 114), although interestingly there are times when Visual Basic 3.0 does not enforce standards

in the Windows 3.x style guide (Microsoft, 1993a). For example, titles for dialog boxes are not required (VBPG 97).

Functional completeness covers the ability to provide functionality at all levels of abstraction required for a system's adopted tasks. Where abstract commands at the functional level of abstraction are largely operations on databases, functional completeness can be readily achieved. There are extensive constructs for information systems (VBPG 453); images can be stored in the database (VBPG 466). There are, however, several features that introduce the risk of losing functional completeness. For example, tasks that require accurate color presentation are obstructed by the use of internal logical palette and system palettes that will produce the 'nearest' match to a color (VBPG 374). This may not be good enough for many applications (not only desk top publishing and image processing, but also Internet applications such as information servers and tele-shopping), even though Windows 3.x itself has extensive palette functions. Some language and environment features can also jeopardize robustness. For example, the DoEvents function must be called to achieve multi-threading, but care must be taken that the procedure which calls it is not called again before the first call returns. If it is called again, then a stack overflow will result (VBPG 417). However unlikely this is, it remains a burden and concern for programmers that would not exist were true multi-threading constructs provided.

The main value of the above analysis is in confirming software architecture as a key determinant of support for properties when developing interactive applications. More classic interactions, e.g. the simplicity-power trade-off between development efficiency and functional completeness, are also exposed by several examples. Functional incompleteness often appears to have been tackled with a local fix that has resulted in inconsistencies between design-time and run time capabilities. The same is true of the equally classic simplicity-efficiency trade-off between development efficiency and run time efficiency, where a chapter of tips and tricks lengthens the developer's coding agenda.

Any global summative evaluation of Visual Basic based on the above property analysis would be misleading. It is hard to trade-off poor support for external properties against, for example, its extensive installation support. There are also clearly development projects where Visual Basic's design-time environment has delivered extensive development efficiency without compromising functional completeness. Even so, it would be a surprise if these developments had particularly adventurous user interfaces, since key external properties are not well supported by Visual Basic.

As with most 'commercial tools', the driving forces in the market concern are internal properties that are foremost in the developer's mind, since all benefits here accrue to the developer. Improvements in external properties

usually accrue to the end-user, with extra costs for the developer that may not be recoverable. This balance of provision for external and internal properties is evident in the site reports in the next section.

5.5 Experiences at Research and Development Sites

The analysis of tools and materials will now be completed by considering broader experiences at four sites, three in Europe, and one in North America. The two development sites develop business critical hardware and software for internal usage and for sale as products. The two research sites develop state-of-the-art prototypes for both internal and external clients.

5.5.1 Development Work at a Large Systems Manufacturer and Integrator

Nature of interfaces

The nature of interfaces designed at this site is not homogeneous: very different kinds of applications are developed. They extend from legacy applications (including the administration of operating systems) to work-flow systems (based on imaging), new PC-based tools, and client-server applications. Thus, some applications have been on the market for many years and now have a large customer base. These applications continue to evolve. Other applications are only bespoke (for a single customer) and may have a relatively short operational period.

Requirements for legacy applications are less demanding with respect to end-user interaction, but complex with respect to *functional completeness*. Requirements from a few other applications, however, have very sophisticated and specific demands concerning user interfaces.

Materials

In such an established development environment, well known, state-of-the-art software engineering materials are used for activities such as problem analysis and requirements/system specification. GUI-related issues have also been addressed. Specific to GUIs are the provision of style guides and of vocabularies for applications, and the integration of 'heuristic evaluations' and walkthroughs into design and quality assurance processes. These address *user interface integratability* and support assessment of robustness properties such as *observability* and *honesty*. For the more specific demands of some applications, the services of a usability laboratory provided by a related research department are available, but are currently given limited use.

Tools

Given the different kinds of applications, a great variety of tools are in use, although evaluation tools are not in noticeable use.

A significantly large number of applications have to run on two target platforms (e.g. Unix and Microsoft Windows) simultaneously. There is a proprietary tool for specification and construction of GUIs which is tailored to this requirement of platform multiplicity. This tool, called DialogBuilder (Siemens Nixdorf, 1994), is used predominantly for most interfaces to new and to legacy applications.

Tools available on the market are used according to individual project needs (target platforms and software to be included, e.g. by other development partners). Visual Basic is used for those applications which have only Microsoft Windows as a target platform and which have demanding interface features not covered by DialogBuilder. In the rare cases when applications require the support of very different platforms, the XVT package (XVT, 1991) is used to address *user interface integratability*. Some interfaces are based on a proprietary alternative to XVT which addressed proprietary legacy GUI platforms very efficiently, especially with respect to run time efficiency.

This site maintains legacy applications originally equipped with complex and rather sophisticated forms-based interfaces that support *reconfigurability* and *access control*. In order to provide GUI versions that are *functionally complete* compared to state-of-the-art interfaces, there is tool support for transforming the original forms definition files into GUI definition files. The resulting GUIs can be reworked manually (if necessary) by the standard tool, DialogBuilder. Such tools improve *development efficiency*.

The deciding factors for selecting tools at this site clearly concern the required multiplicity of platforms. The main goal is reducing *costs for development and maintenance*. Predominant are applications requiring only modest and standardized interface features (e.g. as covered by style guides, allowing for the construction of interfaces by specification). For these applications, *multi-threading* seems to be sufficiently supported by window managers, i.e. between rather than within applications.

Modifiability is often inherited from the original forms-based interfaces although restricted to a pre-defined scope of interface features (e.g. language, novice vs expert). Special requirements have to be met with respect to learnability, and with the coexistence of legacy interface variants with GUIs.

5.5.2 A Campus Research Centre

This site is using Tcl/Tk in conjunction with the TIMES Distributed System Builder (Smith and Newman, 1995) as a 'Rapid Delivery' (RAD)

vehicle for both 'stand-alone' and distributed information systems. Provided that the necessary problem analysis is available with which to 'prime' a system, TIMES and Tcl/Tk can be used to provide rapid implementations of full working systems with approximately a couple of days of effort for a stand-alone system and about a week for a distributed system.

The basic system consists of combinations of 'front-end subsystems' (denoted FESS) and instances of information management subsystems (IMSS). The FESS are mostly written in Tcl/Tk although other available front-ends are also used, for example, browsing tools for the World Wide Web such as Mosaic (Dougherty *et al.*, 1994) and Netscape (Pfaffenberger, 1995). The IMSS are usually TIMES systems (a program written in C with an attached database) but could also be simple files or commercial data management systems. A complete system consists of at least one FESS with at least one IMSS (Smith and Parks, 1995).

Several link constructs are supported for interconnection between the subsystems. They can be accomplished by the front-end subsystem directly reading the file in which the data is stored; more 'sophisticated' alternatives include socket connections, pipes, e-mail and intermediate files.

As described in Section 5.4.2, Tcl/Tk can be used as an effective tool for building systems which deliver a number of the external properties detailed in Chapter 2. However, the use of TIMES as an IMSS considerably extends the degree to which these external properties (and the internal properties proposed in Chapter 3) can be delivered while also reducing the amount of programmer effort required, thus improving both the quality of the user interfaces and *development efficiency*.

Most of the properties are achieved by adopting suitable design goals and then by ensuring that the delivered system meets the design. The design of the tools assists in achieving effective implementations quickly. For instance, interfaces are designed which let users make use of appropriate representations on appropriate devices for both input and output, and thus provide *device* and *representation multiplicity*. Tcl/Tk assists with *representation multiplicity* by providing appropriate widgets. The TIMES IMSS assists in the acceptance of a variety of input formats and the provision of alternative output formats by providing a rich set of translation/parse/search capabilities that can be accessed from the FESS.

Non-preemptiveness is a goal which can be achieved by always allowing the user a choice of actions in the design (the tools have no direct influence on this except that they do not force pre-emptiveness). Similarly, *multi-threading* is not specifically prohibited by the tools and large scale multi-threading has been chosen as an explicit design decision in the interfaces built. In a particular situation a user can choose to browse or carry out a 'what-if' investigation then return to the situation they were in and continue. However, at most points they only have one window visible and

must deliberately leave this to continue. Multi-threading is thus assisted by a stack rather than directly addressed by switchable threads.

The IMSS also assists greatly in providing *observability* and *honesty* by making it easy and quick to perform various tasks: locating and retrieving information that has been stored; changing storage representations without losing existing information; and selecting subsets of existing information. *Access control* is facilitated by requiring an explicit 'publication' of information before it can be observed. The publication mechanism is role-oriented, and thus addresses *human role multiplicity*. Information is only available for use by users acting in an appropriate role. Information can still be made 'publicly' available by creating a 'public search' role which is given automatically to all users of the system. The ability to quickly add new storage and indexing capabilities (both statically, by changing the FESS, and dynamically on user request in the TIMES IMSS) means that *modifiability*, *migratability* and *reconfigurability* can be readily supported where required. This same ability means that it is easy to record enquiries and to construct a 'frequently asked queries' facility with the corresponding 'frequently required answers', a very good example of *I/O re-use*.

By design, *reachability* has been approached in a rather unusual way. The TIMES IMSS will not permit deletions, thus it is not possible to remove history and all previous states of the systems are *observable*. Conversely, no state of the complete system can be reached which would negate history. However, it is always possible to create a subsystem containing only part of the information in the existing system. The IMSS provides assistance in ensuring that all such 'viewpoints' are internally self-consistent. Having defined a new viewpoint that does not contain the record of a particular event or does not contain some particular pieces of information, it is then possible to carry out 'what-if' scenarios using this as a starting point. The results can then be compared with other subsets without any possibility of overall system inconsistency arising.

The TIMES IMSS has been designed to be *portable* and already runs on most Unix platforms plus MAC and MS/DOS PCs. The ability to use a variety of communications tools also means that a system can be configured to make use of existing facilities without, necessarily, needing to port the IMSS and the possibility of migrating functionality from the FESS to the IMSS means that a lightweight 'native' FESS (for example Xterm, e-mail tool or WWW browser) can be used, greatly enhancing portability.

Evaluability is a major design goal, and both the IMSS and the FESS building tools have been produced with this in mind. The GENIE system (the UK Global Environmental Change Data Network Facility; Newman *et al.*, 1995) developed at this site has been configured both to allow users to supply comments and to record interactions. This permits the actual usage of the system to be reviewed at regular intervals, providing both usage

statistics and the ability to identify problems with the interface and with the user's understanding of the functions provided.

The rapid development concept means that *development efficiency* is not a major issue. However, *run time efficiency*, which is often sacrificed for rapid development, is important. The IMSS is designed to facilitate the achievement of rapid response with low computing resource use. In addition, the ability to configure an appropriate distributed system allows existing subsystems to be re-used, where this would minimize resource usage. Tcl/Tk, being interpreted, is not particularly efficient. However, if resource consumption problems or excessively slow performance are observed, it is possible to migrate the computing requirements to the IMSS.

5.5.3 Research Centre for Large Engineering Company

Apart from normal programming languages and many standard utilities, this site uses a number of tools for testing purposes and for prototyping purposes. A few of these are briefly reviewed in order of priority.

Visual Basic

This tool for PC software running under Microsoft Windows is used because of its often high *development efficiency*. The proliferation of third party shareware and commercial custom controls increases the chance of *functional completeness*. Visual Basic provides excellent support for normal user interfaces with standard graphics. Database access is simple. It is very easy to write modeless 'direct manipulation' (of the interface) interfaces due to its underlying event model. However, as noted in Section 5.4.3, there are problems with graphics with 'semantic content', support for animation and dialog control. At this site, these shortcomings have all impacted *development efficiency*, *maintainability*, and especially *functional completeness*, by forcing changes to prototypes because design decisions could not be implemented.

Toolbook

Toolbook is another PC tool for applications running under Microsoft Windows. For many applications it delivers good *development efficiency*, and *representation multiplicity* is addressed by specific support for animations and multimedia. Dialog control is scattered onto localized scripts and thus Toolbook suffers from similar problems as Visual Basic here (e.g. it obstructs analysis of *reachability*, obstructs provision of *pre-emptiveness* when required by application domain or user expertise, and obstructs *maintainability* and *modifiability*).

StartView

This is an Interface Builder set for PCs running under IBM's OS/2. The tool is primarily used for its CUA conformity aiming at *user interface integratability*, but it also delivers good *development efficiency*, making it suitable as a rapid prototyping tool.

5.5.4 A Telecommunications Company

Many of the interactive systems developed at this site have real time requirements and very often deal with very large systems (customer databases running into scores of million of records). Several of them are front-ends to larger systems and often employ standard GUI packages (e.g. Motif). Others deal with network concentrators or reconfigure problematic digital switches. This site largely serves two sets of customers:

- internal: large group of heavy users who exercise discretion and can also get the best out of challenging tools and materials;
- external: larger group, who generally must work with 'industry standards'.

Non-preemptiveness is often felt to be necessary since craft (operators) are not expected to be sophisticated users of the system – and can often be the 'front line'. *Insistence*, *predictability* and *pace tolerance* are all considered vital for these and other users.

Performance monitoring tools are heavily depended on – this can make or break a product. In both dealing with customers (scores of millions) and calls (120 million/day), this site is constantly confronted with problems of 'scale'. While several systems are of course broken down into smaller pieces, there are systems that have to deal with a large number of entities. Hence the critical role of performance monitoring to ensure *pace tolerance* and *run time efficiency*.

Portability is both vital and largely assured thanks to Unix. Every size of Unix box is deployed at virtually all levels. Thus tools work in many places. The organization is too large to establish which tools are in regular use. Common tools and materials include: Unix, Unix tools, the X Window system, OpenWin (the organizational standard for *user interface integratability*), and Motif (some developers' preference). Several UI builders are in use, but there are too many tiny and big UI builders throughout the organization to get any coherent sense of current trends.

The site is largely tool- and process-driven in approach. Tools at times take precedence over process, but in the milieu of large scale software construction (not just interactive systems) process plays a very strong role. A standard deployment cycle is followed that is very similar to the one presented in Chapter 1. There are on-line methodologies, which are heavily consulted.

Evaluation and re-evaluation are performed at various stages. The local development methodology dictates a variety of things and has an impact on other things (choice of tools for example), largely by guidelines of the form: what to do when X happens during stage Y of a project development.

5.6 Conclusions

Tools and materials for developing interactive systems is a vast topic. Analysis here has been restricted only to tools and materials that have a positive or negative interaction with an external or internal property, and ones that are used or produced during the specification and construction phases of development.

Various forms of interaction arise between tools/materials and properties. These forms have direct implications for development activities, determining the extent of work and expertise required from developers, and the phases of development when properties can be considered. The most favorable interaction is *delivery*, which can occur during specification and require little expertise and no further effort in later phases. Less favorable interactions are *proof*, *addressing* and *assessing*, which tend to only occur during specification, require considerable effort and expertise, and require further attention to properties in later phases. These variations in attractiveness are reflected in the site and tool reports (none of which mention proofs), where delivery of internal properties during construction is the predominant form of interaction.

It is clear that tools and materials that are currently used extensively are largely ones that support internal properties during construction. Few tools address external properties. The causes of this situation cannot be established with confidence from the range of examples above, but these do allow some informed speculation as to why current tools are uneven in their support for external and internal properties.

One likely cause is the limited attention given to the quality of the final systems that are produced by tools or incorporate re-usable materials. However, this limited attention may reflect a deeper cause that lies in the nature of interactions between external properties and tools and materials.

External properties can often be delivered or proved during specification, but they must still be preserved during construction. Similarly, some properties can be assessed during specification, but the property must still be preserved during construction, and then re-tested during evaluation. Thus appropriate specification constructs and assessment tools cannot guarantee satisfaction of properties in the final system. This reduces the attractiveness of tools and materials that only interact at these levels with key properties. In contrast, a construction tool that delivers a property does so with no further effort. This difference in the effectiveness of specification and construction phase interactions may be an important cause of uneven tool and

material support for external and internal properties. It cannot however be the sole cause.

Differences of support between specification and construction are wider than they need to be. Construction tools often only assist in achieving properties, even though the gap between assistance and addressing/delivery may not be particularly great, especially when missing supporting constructs have already been implemented. Thus a construction tool that only assists with a property will require far less effort from developers if a relevant supporting construct is implemented and encapsulated. Such a simple addition of capabilities can be called an 'assistance upgrade', and this may be the obvious way to quickly improve on the state-of-the-art.

For most tools, 'assistance upgrades' should generally succeed. However, few have clearly been attempted in recent years. As such upgrades would be technically straightforward, the root cause of slow improvement of tool support for external properties must lie elsewhere.

The most plausible cause relates to the beneficiaries of 'hard' properties (i.e. ones with which proof, delivery or addressing interactions with tools/materials occur). When a property is satisfied, there are various beneficiaries. The major beneficiary from internal properties is the software developer. However, when external properties can be delivered, the users of a system rather than the developers of the tool or material will be the major beneficiary.

Tools and materials that improve internal properties have an immediate benefit for the software developer. Tools and materials that improve external properties have less immediate benefits. Although customer relationships should be strengthened, and the reputation of the developer for quality development should improve, the actual return to many developers on investment could remain uncertain, if not unclear. However, the customer-contractor relationship in software development is not fixed in stone, and closer, more open and more cooperative relationships are developing, just as they have in many areas of manufacturing. Interestingly, the one report from an in-house development site does identify specific external properties that are required by their operators ('craft' in telecommunications speak).

The apparent root cause of unnecessary differences in tool support for external and internal properties can thus be addressed. The site report from a campus research laboratory shows that external properties can be given focused attention when developing software infrastructure. Furthermore, the facilities in tools like TAE+ that have evolved to meet the needs of a large demanding and varied internal user base can also provide reasonable support for external properties. The obstacles to improving tool support at the specification stage for external properties are not largely due to technical obstacles to realizing some form of interaction, but, as has already been noted, due to the unsatisfactory loose ends that have to be addressed during

subsequent construction and evaluation. For these reasons, model-based UI tools have promise that goes beyond the generally cited improvements in development efficiency. Such tools could also preserve external properties that are proved, delivered, addressed, or assessed during specification.

This concludes the informed speculations on the causes of slow improvement of tool support for external properties. The main value of these conjectures is that they identify possible ways forward, especially the value of model-based development tools.

In summary, interactions with software development that were identified in Chapter 3 have been shown to be substantial, in that examples of different strengths of interaction can be readily found for existing tools and materials, but these interactions are diffuse, diverse and lack coherence. Internal properties are currently covered more comprehensively and coherently. Support for external properties is much more piecemeal, due to the risks of 'property erosion' during construction and evaluation. Model-based tools could be developed to address this 'property erosion'. Thus the predominance of interactions with internal properties reflects not fundamentals, but forces operating within software development. Designs for tools and materials that recognise the nature and degree of these forces are most likely to harness or counteract them.

Tools alone will not solve all problems associated with properties. There will still be trade-offs to be made. As with the last example site, methodology is not irrelevant, and it has some effectiveness even with partial tool support. Thus the proper combination of tools and methodology is also an issue that needs to be resolved.

CHAPTER 6

Example: Interface for Air Traffic Controllers

6.1 Introduction

This chapter introduces a single large example of using the properties and architecture which have been discussed in earlier chapters. The example chosen is an Air Traffic Control (ATC) Support System. The concrete meaning of the abstract properties introduced in earlier chapters will be discussed in that context, showing how one property interacts with other properties. The relevance of this to the design is shown by examples, which are followed by a discussion of possible architectures for the ATC Support System.

A realistic example of this kind is inherently complex. Different individuals within the overall system potentially fulfill multiple roles. Both real time and safety-critical aspects of the system need to be considered. In this respect the example provides a challenging test for the merits of the design approach proposed in this book. It also serves to illustrate situations in which considerations of the functional domain may need to over-ride user interface engineering considerations.

6.2 The Air Traffic Service

The global task of Air Traffic Management is provided by individual national Air Traffic Services containing many hundreds of people carrying out support, maintenance, technical, controlling and other activities. The major real time activity in this overall system provides Air Traffic Control services to aircrew (pilots of civil, military and private aircraft) during flight planning and en route.

The main purpose of the Air Traffic Control system is to ensure flight safety: aircraft must be able to fly from take-off to landing without fear of collision. Given this strict constraint the ATC system aims to offer optimal flight paths to the pilots who are the ATC users: aircraft should be able to take off when planned, fly at a speed desired by the operator at an altitude suitable to the desired flight path – all dependent on other traffic, weather conditions and constraints offered by the particular type of aircraft being used.

In order to achieve these goals, the Air Traffic Control Centres involved in the flight planning and en route control of aircraft need to collect considerable quantities of data about each planned flight – time and place of departure, time and place of arrival, proposed route, height, speed etc. Based upon this data the duty Air Traffic Controllers (ATCos) make decisions about the need for route alteration due to planned traffic, informing the pilots when accepting the flight plan before take off.

The most important part of Air Traffic Control is exercised while aircraft are en route. Changes in weather, other aircraft emergencies etc. all have to be dealt with in real time: duty controllers assisted by an ATC Support System monitor the minute-to-minute situation in the air, giving advice and requests to aircrew to avoid conflicts between aircraft flight paths.

The system being considered in the remainder of this chapter is the ATC Support System needed to help the human controllers manage the vast amount of data available and needed at some time or other in order to make optimal decisions while carrying out controlling activities.

The order of presentation of the requirements from the support system takes into account some simplification necessary for the purposes of this example. A design would normally begin by reviewing controllers' activities in an existing system in order to permit a direct comparison to be made between the new design and existing practice. For brevity this is omitted and an abbreviated description provided.

The complete ATC Support System interacts with many individuals fulfilling a variety of different roles. This example concentrates on the tasks performed by and for the en route controllers themselves. In practice, however, each individual controller must adopt many roles over time – sometimes even more than one at once – advising pilots, ensuring safety, communicating with other controllers about air traffic movements etc. These multiple roles are indicated in the description of controllers' tasks given below.

The descriptions given make use of a simple framework frequently employed in software engineering – first describe the goals and then the inputs and outputs available to assist in achieving the goals.

6.2.1 A Controller's Tasks

Flight Management

The basic task of an en route controller (an ATCo) consists of monitoring flights in progress. In an ideal world this would mean checking that every aircraft follows its pre-planned (accepted) route. Controllers, however, have to face perturbations as well as poor planning. As soon as a potential conflict (which may lead to a dangerous situation) occurs, the controller must take action to reroute aircraft to avoid the conflict. In order to do

that, controllers often have to choose an altered (sub-optimal) route. This means that they have to build (or ask the Support System to build) a new subset of routes. Once these are determined to be satisfactory the controller must then undertake the correct actions (e.g. requesting change of course to pilots) to adopt the revised plan.

Control Co-ordination

No one controller can manage all airspace. The world's air space is divided into regions roughly in line with national boundaries or other major geographic division. Within each region the sizes of individual control sectors are designed as far as possible to even out controller load. Each controller on duty monitors all flights originating in, passing through or terminating in the single sector assigned – while the aircrafts are in that sector. The size of control sectors is such that most aircraft in transit (i.e. not merely carrying out local flying) will usually cross several sectors between take-off and landing. If the controller of the originating sector for a flight takes an action to modify the planned route then controllers of all subsequent sectors will need to deal with the modified route. Such changes are, of course, cumulative from sector to sector as modifications are found to be necessary. This introduces a new task for controllers: co-ordination and negotiation with other controllers either in the same Control Centre or in a neighbouring one. To simplify this co-ordination, airspace is generally organized into airways so that flights crossing sector boundaries do so only at pre-defined points. Negotiations between sector controllers are then about the time and altitude of such a crossing rather than its position.

Focusing on this en route Air Traffic Control and supposing that take offs and landings are managed in dedicated sectors (usually referred to as Terminal Control Areas or Terminal Control Zones), it may be assumed for the purposes of this example that a controller's duties consist of these two subtasks: negotiating with other controllers and managing flights within the sector – which includes monitoring and building new trajectories. In practice in a busy Air Traffic Control Centre these two subtasks are often performed by a pair of controllers working together.

6.2.2 Available information

The data used for monitoring airspace is synthesized from different sources. First, flight plans are used: though the static information they provide may become obsolete, they make it possible to make assumptions about the future movements of a plane. Then, radars provide real time information (refreshed every few seconds) about the positions and altitudes of aircrafts in flight. Every aircraft carries a device called a transponder that uniquely identifies it, so that radar information can be correlated with flight plans.

Finally, even though this is still under development at the time of writing, the existence of a data link between every aircraft and the ground will be assumed.

In addition to this information about flights, the system also has knowledge about the geography of the sector: its borders, the airways in it, and a number of pre-defined 'waypoints', used as references to build trajectories and to communicate with crews and other controllers. Finally, the pieces of information entered by a controller when dealing with aircraft may be made available to other controllers during negotiating.

6.2.3 Actions

A controller takes a number of actions when monitoring flights, building new trajectories, or negotiating with other controllers. Some of these actions are internal to the ATC system, and depend on the interface used. Using the interface, the controller also has to perform a number of actions that have an impact on the real world. Compared to the information available, the controller has very few ways of acting on the situation, because every meaningful action is mediated by other humans, especially crews (pilots are not yet prepared to let controllers take actions that affect the state of their aircraft). This is why most actions are in fact communications. The controller communicates with crews, in order to request them to implement the decisions made when solving problems.

The first action to be taken as soon as an aircraft enters the sector is to open and check the communication link. Then, when it is necessary, the controller asks the crew to change speed, altitude or heading. Finally, when the aircraft is about to leave the sector, the controller tells the crew to contact the controller of the next sector, and makes sure that the new communication link is established.

In the same way, the actions involved in coordination with other controllers are basically communications. Negotiations among controllers generally occur when a controller wants to hand over an aircraft to another controller. If this is not performed automatically by the system, the controller has to establish a communication link, name the aircraft and propose a waypoint, a time and an altitude for the transfer. The other controller can then accept, or propose other solutions. As controllers share the same computing system, these pieces of information are easily available, provided that they are entered into the computer.

6.3 A Simplified ATC Support System

Now that the controller's task and the means available have been described, the new Support System must be designed. This is, of course, too large a task to be done completely in this example; many of the details are not

relevant to this book. For these reasons the support system considered will be simplified; details will only be introduced when useful in illustrating points being discussed.

The primary aim of an ATC system is to enable controllers to handle an important amount of air traffic without jeopardizing the safety of aircrafts. For the purpose of this book, it will be assumed that a reasonable way of achieving this aim is to create a single integrated computer supported system which includes the various existing subsystems (flight plans management, radar display, radio links and so on) and provides a uniform, usable interface for the air traffic controllers.

An ideal support system design suggests that the system should:

- accept flight plans electronically and log them automatically;
- establish communication with the pilot when the aircraft crosses into the sector (the controller of the previous sector having both informed the new sector controller of the entry and requested the pilot to establish communication with the new controller; both pilot and ATC system use a time-out mechanism to indicate a problem to the preceding sector controller if successful communication cannot be established);
- make available to the controller in advance the expected times of aircraft entry to and departure from the sector and the planned flight route through the sector;
- provide an information retrieval and computational support system for investigating putative flight plan alterations ('what if' scenarios), using existing flight plans, permitted 'routes', actual situations in the past, simulations of possible revised routes;
- log chosen flight plan revisions and notify the next sector of the projected consequences, if any;
- notify the controller (and perhaps the pilot) if any anticipated event is not detected when expected (e.g. a failure to establish or maintain communication; a failure to respond to a request).

6.4 External Properties

As may easily be imagined, external properties are crucial to the success of an ATC support system. More generally, the relative weighting of the properties depends on the application domain. Air traffic control is an open, cooperative and very safety-critical application. Many people (e.g. controllers, pilots and supervisors) must cooperate for the ATC system to function at all; because of the nature of the system, the overall system safety must be the overriding concern. This influences strongly the way in which the priorities of the desired properties must be balanced; in several cases domain requirements take precedence over 'nice' software properties.

For each property, with the exception of goal completeness, one or more situations are described, where this property is of significant importance in the system to at least one of the many people involved. In a number of cases, overlaps between properties are identified.

Safety requirements

Because of the application domain of the ATC system, safety requirements override almost every other requirement. This means that the *robustness* of the interaction is very important. Robustness means – among other things – that the system tolerates errors and deviations from ‘normal use’. As also noted in subsections below, redundancy in the system and in the dialog assists in deviation detection. In order to be able to tolerate deviation, either the controller, support system or both need to detect potential errors and take appropriate action before the situation gets out of control. The examples noted under observability, insistence and honesty are all cases where the system is tolerating expected deviation of one sort or another. For these reasons, it is essential to examine interaction robustness properties before interaction flexibility properties.

Furthermore, *unexpected* deviation also has to be taken care of. As a general rule, any safety-critical system such as an ATC system, must have anticipated all possible events and have provided actions for each. Yet it is impossible to enumerate all possible events in most real systems – they are usually countable though infinite. This impasse is normally overcome by first defining a *safe default action* which will occur whenever any otherwise unanticipated event occurs (‘unanticipated’ means that the event is not covered by a specified set of conditions). Later on, as unanticipated problems are observed, explicit tests and responses are added for these specific possibilities.

This method of coping with deviations could be described as Deviation Intolerance since, unless a specific case has been identified, in which case it is not a ‘deviation’, then a specified action will always be taken. Conversely, it could be thought of as Deviation Tolerance, since all user actions lead to inherently safe operation (but not, necessarily, to what the user would have desired).

Further discussion of safety aspects is found below in several of the subsections on individual robustness properties.

6.4.1 Goal completeness

The principal purpose of the ATC system as a whole is the safe passage of aircrafts from take-off to landing with minimum disruption to the original flight plan and minimum extra cost for operators as secondary aims. The goals of individual humans using the air traffic control system may or may

not be related to these purposes. However, the ATC support system will be complete insofar as it enables the controllers interacting with it to achieve the overall purposes and discourages them from taking actions that are inimical to those purposes. A full judgement on goal completeness is only possible when the system is in operation. It may even be necessary to wait until after the system is taken out of service before the final assessment can be made.

Basically, a task-analytic approach to completeness would not suffice in this safety-critical system. The task analysis results in a task model, and task completeness is demonstrated by showing that all adopted tasks from the model are covered. As pointed out above, however, the real world environment interferes with the ATC system in such a critical manner that task completeness cannot guarantee the safety of the system.

6.4.2 Interaction Robustness

Observability

According to its definition, observability means that a system must make all information that is relevant to the user perceivable. The key word here is 'relevant', especially if information overload is to be avoided: relevance depends on the situation and time. A differentiation can be made here between the capability of observing (i.e. making information available on request) and providing information automatically. Forcing something to be observable may lead to clutter and to important things being overlooked.

An alternative to basing relevance on the user's current tasks is to allow the user to perceive anything he or she can name. 'Name' should be interpreted as meaning 'provide a description for'. If the description is unambiguous then the required information should be provided. If not, the user should be given information about the choices of things that could be made available, and which the designer expected could be useful in the current state.

Some information can be displayed continuously, because it is essential and/or because its display is inexpensive in terms of perceptive load. For instance, the current positions of airplanes are always represented on the display screen because they are essential for managing air traffic. Similarly, the past positions of airplanes over the last minute or so are also represented, because they give a readily assimilable indication of direction and speed, and because they can easily be integrated in the display.

Other kinds of information do not merit continuous display, but need to be observable. This can be accommodated by always making search or browse facilities available. This would allow an ATCo to examine any of the information in the system if desired, since anything known to the system which relates to aircraft or to pilots or to the system itself or to other

air traffic controllers is, potentially, relevant to the ATCo in particular circumstances. For instance, the ATCo may wish to check the route that the pilot has entered in the flight management system of the airplane. Observability will allow the ATCo to detect blatant errors in the route, and even to decide whether the pilot will need special care and guidance, because he does not seem to know the area well enough.

The trade-off between observability and browsability (or discoverability) – i.e. searching – can be thought of as the trade-off between providing very focused information, specifically oriented to the current task, and providing additional information because it might be useful. The former strategy may lead to essential information not being available when it is needed, the latter to clutter and information overload. Since in any real system, all the information cannot be observable at all times, it is always possible that the controller will need to ‘search’ for the information they require. If a very focused task-oriented strategy is adopted for what is made observable, it will be necessary to provide a browse/search mechanism which allows users to quickly find the information they need.

Finally, awareness (or feedthrough) is a special case of observability, as indicated in Section 2.4.1. It is also important in an ATC system. Suppose that a controller has a potential emergency and wants to redirect an aircraft to another sector, but that sector is currently overloaded. If the system has kept the ATCo aware of conditions in the surrounding sectors (perhaps by varying the color of the boundary with the other section displayed on the screen – green might mean lightly loaded, white average load, orange heavy load) then the ATCo is in a position to make an informed choice without needing to search for information.

Conversely, once the potential emergency is notified to the Air Traffic Control system, it must (insistently) be made observable to the ATCos in the surrounding sections (perhaps by making their boundary displays red). In the emergency situation, it is essential to ensure that these ATCos are aware that the emergency exists so that they can be considering whether there is any way in which they can offer help. In this case observability is not enough. Insistence is required in that the system needs to ensure that the information has been observed by obtaining a positive feedback from each of the other ATCos (the only situation where this would not be the case would be where one of the other ATCos was already dealing with an emergency of equal or greater magnitude).

Access Control

As discussed in Chapter 2, there are two different aspects to Access Control. Firstly, it is intended to prevent users deliberately viewing or changing information that the owner of the information did not wish them to be able to access. Secondly, it is there to make it easier for the controller

to use the system by eliminating irrelevant information and/or unnecessary functionality. The latter aspect of access control is closely related to both observability (users should not have to observe information that is not relevant to their role) and honesty (functions should not be offered if they cannot be used). Putting this in a more positive way, the chance of accidental mistakes will be minimized if users are only able to access those items that are relevant to their current task, and are only able to carry out the operations that are appropriate when performing the task (i.e. this form of access control offers role-oriented support for the user).

In the ATC system, an ATCo would not normally need to access information about aircraft which are not currently in their sector and which are not scheduled to enter their sector. Thus the Access Control mechanism should prevent them from getting this information even if they make a search request that it would apparently satisfy (e.g. the access control mechanism should attach to the request 'tell me about aircraft that are scheduled to land in Toulouse at 1400Z' the condition 'an aircraft is only relevant if it is either in my sector now or it is scheduled to pass through it between now and 1400Z'). Similarly, and even more importantly, an ATCo should not be able to make changes to the information relating to aircrafts that ATCo is not controlling. Thus a request to amend the route for an aircraft, which is specified by referring to the aircraft's identification number, must be rejected, unless it is under the control of the ATCo at this moment (irrespective of the fact that it may be currently in the air and known to the system, or was recently being controlled by this ATCo). However, in some circumstances, it must permit an ATCo to change roles and then be able to access the information that was previously unobtainable, e.g. if a colleague becomes ill on duty, aircrafts may need to be reassigned.

Insistence

A system is insistent if feedback to the user is sustained until some specified user reaction is forthcoming. Note that there is an inherent conflict between the robustness property of insistence and the flexibility property of pre-emptiveness.

In this air traffic control example, an alert is said to occur whenever something moves outside pre-defined limits possibly specified by the procedures or by the ATCo. System alert should always be insistent, as mentioned above for observability.

Suppose that an aircraft that is due in the sector has not arrived: communications have not been opened and there is no trace on the radar for that aircraft, although the previous sector has passed on information of expected arrival. This could mean that the system in the previous sector did not notify a change in plan. The system would then need to notify the ATCo and log a possible system fault condition so that corrective actions

can be taken. In this case insistence operates on at least two different controllers on different time scales. The ATCo needs to 'find' the aircraft, and the supervisor needs to review the operation of the system. If the ATCo does not respond within a prespecified time alert priority is raised. How this is done depends on the alert and preferences – e.g. a klaxon sounds, or the room supervisor is warned of a possible need to take over. As concerns the supervisor, the error message will be stored and reminded every day, until it is handled.

From time to time the controller may need to temporarily suppress a message in order to deal with a more urgent/dangerous situation. This, however, must only be possible within context-dependent limits, if safety criteria are met. Automatic transfer of 'insistent' messages to other people may be one design solution. Conversely, in an emergency, signals that ordinarily might be insistent may need to be suppressed to allow the ATCo to concentrate on the emergency. There is no need to notify the ATCo of the normal arrival in their sector of an aircraft if it is on time, its flight plan does not overlap with the airspace involved in the emergency, and it requires no ATCo action to allow it to proceed.

Honesty

A system is called honest if representations of internal state elements are designed to be interpreted correctly. Honesty is a fundamental feature of all aspects of the ATC system. For instance, it is hardly acceptable for the system to display the position of an aircraft that did not exist; this is why so much effort is still devoted to the elaboration of algorithms for radar signal processing. The design of the support system can facilitate the detection of accidental dishonesty (e.g. due to system malfunction or user misconception) by providing a high degree of redundant information. This was illustrated in the previous section, where the transfer of an aircraft from one sector to another is accompanied by a message exchange between the systems in the two sectors, a display of the incoming aircraft position on the receiving controller's console and the opening of a communication channel between the aircraft and the ATCo. Various controllers should thus each be able to distinguish between a single device/subsystem failure and a genuine 'problem event' and not be confused by thinking that the system was failing to be 'honest', i.e. failing to make the relevant information available.

Pace tolerance

The definition of a pace tolerant system is a system where the user may control the pace of interaction. Since the Air Traffic Control Support System has to operate in real time, it is not pace tolerant in the way a text editor or some computer games could be. A text editor is generally purely reactive,

and users chose their own pace for typing. Computer action games involve real time, but most of them provide a 'pause' or a 'slow speed' function in order to provide a form of pace tolerance. But an Air Traffic Control system cannot offer such a facility. However, pace tolerance can be introduced in secondary functions. For instance, the system allows the ATCo to tailor the length of time-outs such as the one given in the insistence example above (the missing aircraft), but only within pre-specified limits in order not to compromise safety.

Pace tolerance must take precedence over 'temporal' honesty, for example, in busy periods where an ATCo receives many data and voice messages from other controllers and pilots. If the system is to be 'honest' about time, it must present messages immediately as they arrive. Honesty here must be relaxed by preserving the sequence in which they occur 'in reality'.

Predictability

In a predictable system, users can know its behavior from their knowledge of past interactions and current state. As in most other systems, predictability is, like honesty, a major requirement for the ATC Support System. The system must provide the information that is requested when it is needed and it must carry out requested operations reliably and within an expected time period. For instance, when sending a message to a flight crew, the controller will want that message to be delivered within a well-known amount of time. Or, when filtering useless information away from the controller, the system should make similar decisions for similar situations. Also, as for honesty, the provision of redundancy helps the controller detect potential malfunctions and distinguish these from a simple lack of predictability.

Deviation tolerance

A system is called deviation tolerant if it supports the correction of slips and mistakes. This implies that the system is able to detect 'unwanted' events, such events being sometimes that the user failed to act. This also implies that the system is able to take some appropriate action. This could be to 'take control' (as a kind of forced migration of control), or it may involve presenting warning information to the user that deviated or to other users.

Being a safety-critical system, the support system cannot offer much deviation tolerance for the contents of actions related to real time activities. Deviation tolerance, however, should at least be offered for the structure and the sequencing of actions.

For example, the ATCo fails to respond to a routine notification of the arrival of an aircraft, which is on time and for which there is no anticipated problem; since the ATCo could have suppressed the notification anyway, there is no sense in insisting on a response or of immediately notifying

anyone else. However, the failure to respond is an indication of a potential problem, which should be logged and should 'sensitize' the system to look for the possibility that the ATCo is 'malfunctioning' (e.g. the ATC system might reduce tolerance limits on other expected actions by that ATCo – if a record is kept of the 'normal' time, and the variance on that time, taken by that individual controller to respond to each sort of notification then the tolerance limits might normally be at a 95% level, the maximum time needed to contain 95% of actual responses by that ATCo, but after one failure to respond within the required period, the tolerance might be reduced to a 75% level until there have been, say, three successful responses; a second 'slow' response might cause the tolerance to be reduced to the 50% level and a third failure sets alarm bells ringing, literally, asking the supervisor to go and investigate).

6.4.3 Interaction Flexibility

Device multiplicity

Data is provided to an ATC Support System from many devices and several devices are used to interact with the controllers. ATCos are provided with displays for graphical radar output (or synthesized aircraft position displays), secondary displays for routine information, telephone and radio for communications input and output, sound output for audible warnings, pointing devices for direct manipulation input of commands, and even keyboards if really necessary.

Multi-device capability means that the controller is offered a choice between several modalities of renderings in a number of cases. The key to success that has been noted earlier is that redundancy is essential if potential error situations are to be detected and prevented from becoming actual errors. Multi-device capability is needed to provide the communications redundancy: it is no use having many different possible ways of detecting errors internally, if at the end all messages from the system are displayed on a single screen, and there is no other method of getting a message to the ATCo. Conversely, however many facilities a system could offer a user, if all of them must be chosen using a single input device (be it a mouse or a touch screen or a keyboard) then the system is useless if that input device is broken.

Therefore, multi-device capability in the ATC is needed for safety reasons, not only for operators' convenience. The design must include multi-device capability to ensure that the ATCo can do the job even in the presence of some (hardware) faults.

Representation multiplicity

The discussion in the previous subsection indicates that a 'safe' system (one that is fault tolerant in that it is not dependent on a single output device to present its results) must exhibit multi-channel capability. Also, representation multiplicity adds to the safety, but may make the system more convenient to the user. In a number of cases, the ATCo should be allowed a choice between graphical, tabulated or audible presentation of output.

Consider the following scenario. The controller will need to have information about aircrafts' current positions presented geographically on the display, so as to reason geometrically about the situation. On this radar screen there is not enough room for detailed information about aircrafts, such as the time at which they are expected at a certain waypoint. The ATCo probably also wants that kind of data, and this is why another representation of aircrafts is provided at the same time: card flight strips (current systems) or electronic flight strips (systems now being designed) hold that information in a tabular form.

If both device and representation multiplicity are present, all the information about an object does not need to be presented in one way on one device. For instance, the event of an aircraft entering a sector can be represented as a change of color of its representation of that aircraft, and can also be represented as a brief sound. The use of multiple devices for multiple representations can thus take advantage of the different qualities of different devices.

Human role multiplicity

The ATC Support System must support many different controllers, supervisors, etc. Even a single controller, however, may take several differing roles over a period of time (e.g. establishing communication with an aircraft, performing a 'what-if' simulation, passing information to another ATCo). Indeed, as far as the ATC system is concerned, the ATCo may be performing several roles at the same time if the system supports multithreading, since this permits several roles to logically coexist even though the ATCo can only actually be interacting with one of them at a particular instant.

As has been discussed in the earlier subsections on observability and access control the role which the system ascribes to a particular interaction thread with the ATCo will control what information is made observable and what information can be retrieved (by searching or browsing). It may also change the functionality, which is provided by the system to the ATCo.

On occasions, a particular ATCo will have to perform roles outside his or her normal duties. There is thus a close relationship between this form of human role multiplicity and customizability. On the other hand, where different humans are performing similar but not identical roles, effective

support for human role multiplicity might best be achieved by providing adequate reconfigurability.

Output or input re-use

The second purpose of an ATC Support System, after security, is to enable controllers to manage air traffic efficiently. Therefore, every service offered by the system to save time is very useful. I/O re-use is such a service. Earlier input can be re-used, because an earlier situation reappears, or some output should be forwarded to another controller or to a pilot. Two examples follow.

When an airplane moves from A's sector to B's sector, the relevant flight information is presented to controller B. Later, when the airplane moves on to C's sector, B wants to re-use the flight information to send it to C.

When an alarm occurs on the display, the ATCo shall re-use the output as input to send it to all other relevant ATCos and pilots.

Multithreading

Multithreading means that the user can execute several tasks at a time. Air traffic control is a very demanding activity in terms of multithreading. Every controller receives input from a variety of sources including radar systems while communicating with other controllers and pilots. A controller may be simultaneously working on one or more information searches and what-if simulations, as well as making amendments to existing plans and filing revised flight plans.

For instance, suppose that an ATCo is using the radar display to build a new set of routes for conflicting airplanes. If a pilot calls during that activity, what the controller wants to do is start a new thread of actions in order to store the information given by the pilot, identify the possible new problems it causes, and begin to deal with it. Depending on whether the system provides multithreading or not, the ATCo will be able to start that new thread without losing his previous unfinished work, or will have to choose between losing it or deferring the handling of the call to a later time.

The ability of a system to support browsing or searching for relevant information, referred to earlier in the discussion on observability, could provide another example of multithreading. If the user must explicitly stop the current task and switch to an information-seeking task which must be completed, or abandoned, before the original task can be resumed, then the system is providing only a single thread. If, on the other hand, a user can initiate a search or browse through the information space while retaining the ability to continue with the 'current' task, then multithreading is being provided.

Providing multithreading would also facilitate achieving the next goal,

non-preemptiveness. If multithreading is available then, in any situation where the system requests a response, the user can be allowed to simultaneously choose one or more of the following:

- reply as requested;
- request further information to assist in choosing the response;
- start a different ‘conversation’, e.g. a replanning simulation exercise (a ‘what if’).

For example, an ATCo could start a replanning simulation exercise, then look for more information, and finally provide the response as requested.

Non-Preemptiveness

Complete non-preemptiveness means that the system must tolerate any permissible event occurring at any time, whether that event be due to user action, communication from another system or communication from another component of the same system (e.g. the radar). Since there are many users and many subsystems, who all perceive themselves as operating independently, it is obvious that an action by one user cannot be synchronized with actions of other users. Furthermore, if the ATCo is to be allowed to ask for further information at any point (see the paragraph on ‘honesty’, above) and to carry out ‘what-if’ simulations, then it is necessarily the case that no interaction between one ATCo and the system can be permitted to be pre-emptive.

Reachability

Reachability means that any perceivable state of the system can be attained from any other perceivable state. This property cannot be met in its pure form in air traffic control systems, because ATC is a real time task that models the real world, and some events are irreversible in the real world. This should be reflected in ‘future reachability’ states in the system. Thus, if an aircraft lands, it is not sensible to want to reach states which represent it as still being in the air when considering ‘what-if’ scenarios related to the immediate future. However, if the aircraft lands badly, it may be important to consider what went wrong during the descent. In this case it is essential to be able to reach the historical record of the actual states of the system during the descent (this will allow the performance of the system and of the controller to be reviewed as well as that of the pilot).

Reachability should be provided as often as possible, especially in non-real time interactions. As noted above, reachability is given essential support by ‘history’, even where things have not gone wrong. For instance, if the ATCo has discarded a flight strip, and another ATCo asks questions about that flight, the ATCo will want to retrieve the flight strip. In cur-

rent systems (with card strips), this is done by searching the waste paper basket.

'Full' reachability would let the user move from any perceivable state to any other state, and this should *not* be possible in the ATC Support System for safety reasons. Firstly, only states corresponding to the real world should be reachable. Secondly, system states representing 'dangerous situations' or conflicts must not be reachable (or only when special conditions are met), e.g. a warning should be issued if the controller attempts to simulate a flight replanning in which two aircrafts are flying with too small a separation.

Reconfigurability

Reconfigurability means that the ATCo may change representations and operations. The ATC Support System will be used by many different persons and under many different circumstances. Therefore it should allow for a certain degree of reconfigurability, and a number of examples of this are given above (changing time-out times, altering limits, changing defaults etc.). Each user must be allowed to customize the ATC Support System for their own use on a short term basis (e.g. for this session) or on a longer term.

Another type of reconfigurability is necessary when changing over, especially in evenings. A control sector is usually attended by two or three controllers, but at night a controller alone is enough: there are few enough aircraft that flight managing and coordination can be performed by a single person. This means that displays and input devices have to be reconfigured so as to allow one person to do the work of two. For instance an ATCo will not want to switch from one mouse to another every time he or she has to interact through a console that is usually used by another ATCo. The mouse on this console would then need to be reconfigured from 'one display only' operation to 'all displays' operation.

Adaptivity

Adaptivity means that the system changes representation and accessible operations as a result of surveying the situation and the ATCo's interaction pattern. For instance, the system may be able to know that some group of aircrafts poses more problems than other aircrafts in the sector, because the ATCo keeps interacting with their representation. The support system could then provide a more visible representation for these aircrafts, e.g. by using a brighter color.

This kind of adaptation very easily conflicts with both honesty and predictability, because it changes the behavior of the system. During the operation of a safety-critical system like the ATC, adaptation must be done with extreme care and – when used – with full explanation to the ATCo.

Migratability

The ATC possesses migratability if the ATCo can ask the system to automatically start some tasks usually initiated by the ATCo and, vice versa, if the ATCo can 'take control' over some tasks, which the system used to perform automatically.

The system will have a number of default actions which under normal conditions are sensible and safe actions. But under extreme conditions (heavy load or dangerous situations) the ATCo must be able to 'take control' and possibly change the default action to something else.

For instance, algorithms are being developed to automatically resolve conflicts between aircraft flight paths. These algorithms are still limited, but can be useful to alleviate the ATCo's load. Then the question is how to split the task between the ATCo and the computer. Migratability is important here. For example, a solution would consist in the system deciding that the ATCo is overloaded (adaptation) and proposing to take over. Then, if the ATCo accepts, he or she will want to be able to take over again, either by telling the system to stop solving that problem or by imposing his or her choices over the system's.

6.5 Applying the PAC-Amodeus Model

This section presents a software architecture for the Air Traffic Control Support System. Consider the external specifications that have been used for designing the software architecture. The system supports two workstations per sector. Figure 6.1 schematically presents the user interfaces of the displays of the two workstations. The monitoring tasks supported by each workstation are complementary and performed by two air traffic controllers.

On the first workstation the radar display is not editable. The controller can modify information about an aircraft by using the tool palette, located on the left of the radar screen.

On the second workstation, the controller can modify the flight path of an aircraft on the radar display by direct manipulation. Furthermore the controller can obtain information about an aircraft by selecting it. Consequently visual consistency (see representation multiplicity property described in Section 2.3.2) of the two workstation displays must be maintained (except when one of the controllers is doing some replanning simulation).

Figure 6.2 shows one possible PAC-Amodeus software architecture for implementing the proposed Air Traffic Control system. Figure 6.3 presents the software agents' hierarchy organizing the Dialog Controller.

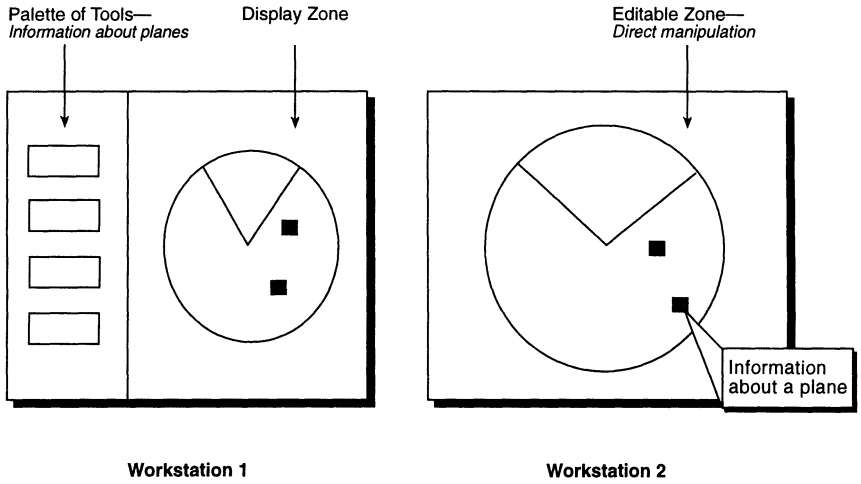


Figure 6.1 *External specification of the two workstations belonging to one sector. Each display shows a radar picture.*

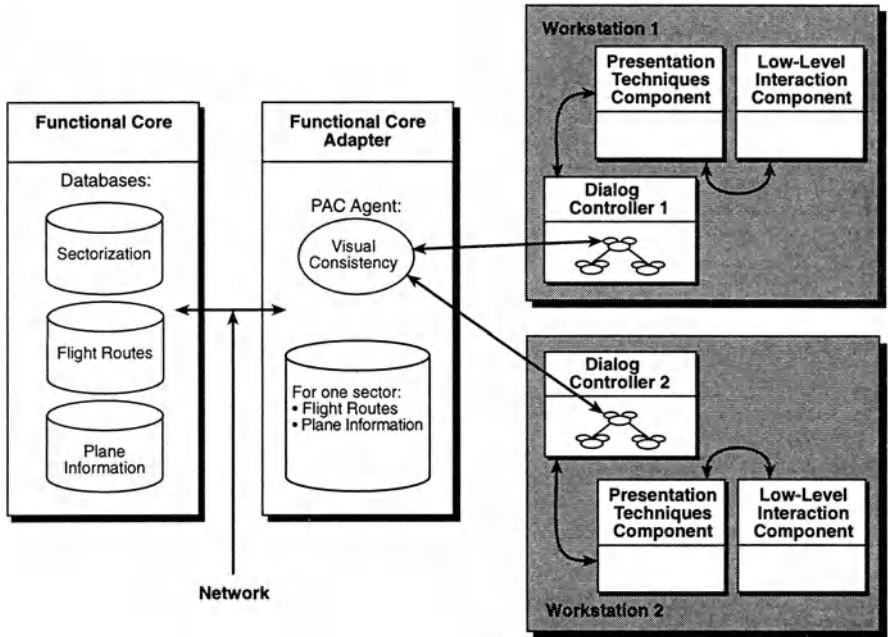


Figure 6.2 *Software components implementing the Air Traffic Control Support System (applying the PAC-Amodeus model).*

At the right-hand side of Figure 6.2, the Low-level Interaction Component (LLIC) denotes the underlying software and hardware platforms. It receives mouse and keyboard events from the user. It also manages the presentation and contains functions to display the radar picture inside a window. The Presentation Techniques Component (PTC) bridges the gap between the Dialog Controller and the LLIC. Neither Dialog Controller (DC) depends on the functions displaying the radar screen, for example.

At the left-hand side of the picture, the Functional Core (FC) maintains and manages the database. The database contains information about:

- sectorization
- flight paths
- aircraft information.

The database is linked to the workstations (two per sector) through the network. To enhance the run time efficiency on each workstation, the information about aircrafts and routes of one sector is duplicated and stored in the Functional Core Adapter (FCA). This option guarantees the stability of the response time because no request is sent through the network. On the other hand this option will increase the number of messages through the network to update the duplicated databases. Moreover the FCA provides for communication between its two adjacent components (i.e. FC and DC) by implementing a communication protocol. It is therefore possible to receive information through the network and to handle user events. This will be managed within the DC, which will be passed network information by the FCA, and receive events from the presentation techniques components.

The hierarchy of PAC agents organizing the DC is presented in Figure 6.3. A PAC agent is composed of three parts (see Chapter 4):

- the abstraction facet
- the control facet
- the presentation facet.

The Dialog Controller (DC) is comprised of PAC agents. There is one DC and thus one hierarchy of PAC agents on each workstation. Figure 6.3 shows the two hierarchies. A dedicated agent within the FCA maintains the visual consistency of the two workstation displays, and is thus linked to the two PAC agent hierarchies.

- The root agents 'Root1' and 'Root2' are in charge of the global control of the interaction with the users. The Presentation facet of each root manages high-level layout and displays ornaments such as frames and separators. The abstraction of each root facet receives information about the flights it should display.
- The Radar agents synthesize the radar pictures. The abstraction parts receive the flight information from the root agent. On workstation 1,

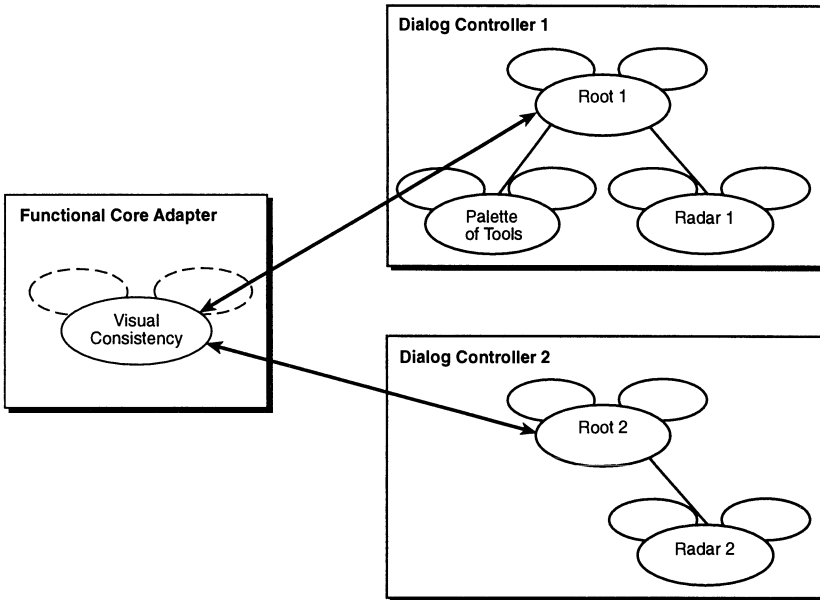


Figure 6.3 *PAC agents organizing the two Dialog Controller components.*

the Radar 1 agent only displays the information. On workstation 2, the Radar 2 agent is more complex and handles the interaction with the user within its zone on screen.

- The 'Palette' agent corresponds to the palette of tools depicted in Figure 6.1.

Conclusions

7.1 Predictable Quality?

The work on this book began with the ambitious aim of forging links between the external and internal aspects of software quality for interactive systems. To address this aim, the Working Group 2.7(13.4) adopted the strategy of associating quality factors with software phenomena. As long as the associations are valid and significant, quality can be addressed via the development process and the tools and materials that support it, rather than by well-intentioned but precarious efforts of highly skilled individuals who invariably lose contact at some point in its life cycle.

The strategy of associating quality factors with software phenomena can be summarized as follows.

- Quality factors have been expressed as external and internal properties.
- Software phenomena have been addressed as the use of methods, architectures, tools and materials within a structured and well managed development process.
- Properties have been associated with selected software phenomena (architectures, tools and materials) by demonstrating interactions between the two.

In the process, it has been shown that quality can be addressed, and in parts proven or delivered, by the judicious use of software architectures, tools and materials. The goal of this book, which is to describe relationships between the process of software construction and system quality from the users' perspective, has therefore been met. We have been able to achieve this in breadth, with some supporting detailed analyses.

The main contribution of the book is to establish a conceptual framework that supports and guides analysis of interactions between software properties and software phenomena. This framework has been exercised in three important ways: by applying it to the currently critical area of software architecture; by applying it to a wide range of tools and materials, with three in-depth analyses of commercial tools, and some broader site reports; by applying it to a demanding application area that is safety-critical, involves multiple cooperating users and has real time requirements. Taken together, this is a comprehensive initial validation of the conceptual framework. The framework has been shown to be applicable to key aspects of

software development and demanding facets of application design. Working Group 2.7 believes that more extensive in-depth analysis can be performed within the framework developed above.

The framework has strong predictive potential for analysis of systems, architectures, tools and components. This potential was demonstrated by the consistently broad range of insights that were yielded by the analysis of existing architectures and tools. The next key step is to go beyond potential to proven effectiveness. To do this, it is necessary to complete the scientific process as follows.

- All identified interactions between software properties and phenomena must be validated by showing that the interaction holds in a representative range of practical scenarios.
- The significance of all identified interactions must be assessed, i.e., what should the real rewards (or costs) of each interaction be?
- All proposed rewards (or costs) must be validated by showing that they arise in a range of practical scenarios.

Working Group 2.7's work has reached the point where it is possible to address the more practical question of significance. Granted that these interactions exist, then what are the implications? Are they imperceptible in practice, or a mild nuisance, a moderate concern, a major impedance or facilitator – or an absolute make or break? They are all of these, some of the time.

Once the significance of each interaction has been understood, it is possible to set out to test the validity of the proposed implication: does the interaction really have the identified implication? There will be some very difficult experiments to design once credible hypotheses can be made about the significance of interactions.

When the scientific process yields results, these must be finally exploited by an engineering step:

- Effective methods and tools must be developed that deliver the rewards of positive interactions and avoid the cost of negative interactions.

This is the hardest step of all. Even when there is scientific validation of the practical significance of interactions between properties and software phenomena such development is far from easy. It expresses in fact the long term goal of the work of Working Group 2.7, that is, to let quality be addressed via the development process and the tools and materials that support it, instead of via well-intentioned but precarious efforts of highly skilled individuals. The process and practice of developing interactive systems needs to change so that high quality systems could reliably be developed from the perspectives of both endusers and developers.

Figure 7.1 summarizes the process of analysis, validation and exploitation. There are six steps in this process (where the first step comprises two parallel substeps).

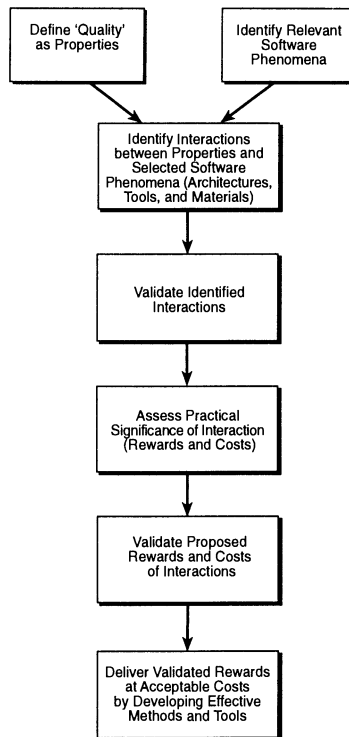


Figure 7.1 *Strategy for Effective Development of High Quality Interactive Systems.*

7.2 Contributions

We feel that we have made good progress on the first steps towards our vision of predictably effective software development for interactive systems. This claim may be justified on the basis of a thought experiment that considers how well the group has done on each of the (sub)steps from identification of properties/phenomena to the near-inevitable attainment of quality.

The (sub)steps on the route were shown in Figure 7.1. In summary, they are:

- identification of external and internal properties (substep);
- identification of relevant software phenomena (substep);
- identification of interactions between properties and phenomena;
- validation of interactions between properties and phenomena;
- determination of practical significance of interactions between properties and phenomena;

- validation of the significance of interactions between properties and phenomena;
- effective exploitation of significant interactions between properties and phenomena.

For each step, we could ask how well we have done in the work reported above. We could then ask what the probable attainment for the step would be if the best work in the world could be synthesized and condensed into a single (though very lengthy!) report. Lastly, we could ask what our likely acceleration for the step would be if Working Group 2.7 spent another year on this report (with assistance from any expert we cared to name). We have done this for each (sub)step, but prefer to leave readers to form their own judgements. However, we will conclude by summarizing the more significant answers to some questions for each (sub)step.

7.2.1 Identification of external and internal properties

The work described in this step is an answer to the question: ‘Which properties encapsulate desirable properties of an interactive system’s external behavior or internal structure?’ Chapters 2 and 3 presented the answer.

Our set of properties is the first to span a range of both external and internal aspects of quality. In the process, we may have gone slightly beyond the best synthesis that would otherwise be available, see, e.g., Nielsen (1994) and Gram (1995). Still, we have omitted several key external and internal properties, and given this knowledge, we could address these deficiencies (e.g., we have omitted ‘learnability’ and ‘guessability’ (Jordan *et al.*, 1991) as external properties).

7.2.2 Identification of relevant software phenomena

The work done in this area is an answer to the question: ‘What are the key methods, techniques, tools, structures and components for software development?’ Chapter 3 started to give the answer, which was then extended in Chapters 4 and 5.

The discussions in these chapters cover several aspects of methods, techniques and tools, but there are many software phenomena that are not covered well (e.g., development methodology and process), and thus more could have done here. However, the field of software engineering is well documented, and thus with more work we could catch up with the current state-of-the-art.

7.2.3 Identification of interactions between properties and phenomena

This gives an answer to the question: ‘What are the positive, neutral, and negative interactions between software quality and software phenomena,

and what must developers do to exploit positive interactions?’ Chapters 4 and 5 presented answers for selected software phenomena (architectural models, tools and materials for specification and construction), but represent only a first attempt to explore and describe systematically interactions between quality and software phenomena. We feel that it is a worthwhile although limited contribution for this step. Extending the analysis of interactions seems to be straightforward, so that the coverage of interactions in Chapters 3 to 6 should be readily extendible.

7.2.4 Validation of interactions

This area of work attempts to answer the question: ‘Do the identified positive, neutral and negative interactions between software quality and software phenomena arise, as predicted, for real tools, at real development sites, and for real application developments?’

Chapters 4 to 6 present our partial answer. Some selected interactions were discussed on a few focused examples. But much more supporting evidence could be gathered from the wide-ranging experience of other software developers.

7.2.5 Significance of interactions

A thorough analysis of the significance of the interactions would answer the question: ‘What is the likely effect, in terms of development costs and benefits, as well as gains for users, of the identified positive interactions between software quality and software phenomena?’

We have not addressed this question systematically, although there are comments relevant to this question at several points in Chapters 4 to 6. But even the very piecemeal analysis here constitutes an improvement on the current state-of-the-art, because very little work of this nature has been reported so far.

7.2.6 Validation of significance of interactions between properties and phenomena

To validate the significance of interactions means to answer the question: ‘What is the real effect, for real application developments, in terms of actual development costs and benefits, as well as measured gains for users, of the identified positive interactions between software quality and software phenomena?’

No such validation has been undertaken, because it requires substantial work and experimentation. But it is hoped that the framework presented in the book may be followed up by research groups that use it as a basis for carrying out individual experiments and studies.

7.2.7 Effective exploitation of significant interactions between properties and phenomena

Work here would answer the question: ‘Where are the methodologies and supporting tools that will deliver systems with a required property profile?’ To the best of our knowledge, such methods and tools do not exist and will not exist for the foreseeable future.

The lack of a good science base for designing CASE tools means that while there are many tools which are effective for many aspects of many development tasks, and which support some external properties for users, progress on this step is piecemeal and unpredictable. As a result, tools and methods can often be developed that improve some aspects of internal and external quality, but make others worse. With a better understanding of key properties and their interactions with architectures, tools and materials, such undesirable setbacks should not arise.

7.3 Epilog

The work reported here has largely been one of conceptual ploughing. The field formed by the intersection of Human–Computer Interaction and Software Engineering has turned out to be very large, and Working Group 2.7 has not been able to plough all of it as evenly or as deeply as we had originally hoped. However, a wide range of individual experiences and those reported by colleagues in their publications have combined to reinforce our view that we have been ploughing in the right direction. Our hope is that the field has been ploughed enough for many seeds to take root and flourish.

APPENDIX A

Glossary

This glossary comprises a full lot of the terms used in the book in alphabetical order, with explanations or definitions in each case. The numbers in brackets [...] indicate in which section each term is defined.

Access Control: information access can be controled, depending on the role of the user. [2.4]

Adaptivity: the system can initiate customization of the interaction. [2.3]

Architectural model: Abstract model of an interactive system. [3.3]

Arch/Slinky: architectural metamodel for interactive systems. [4.2]

Articulation: a sequence of user actions to communicate a command. [1.2]

Binding Services: tools to construct the interactive system from modules. [1.5]

Client: person assessing the scope of a project. [1.4]

Coded module: the software piece which together with other modules implements the interactive system. [5.1]

Coding/Module Construction: implementation and debugging of modules. [1.3]

Cognitive walkthrough: inspection of a design by specialists looking for learning problems. [3.3]

Command: a single user action at the functional level. [1.2]

Construction tool: tool used to transform requirements and specifications into coded modules. [5.1]

Development Efficiency: a measure for how efficient resources are used during design and construction. [3.2]

Deviation Tolerance: the system supports correction of slips and errors. [2.4]

Device Multiplicity: the system offers several communication channels for input and output. [2.3]

Dialog (D): the software component controlling task sequencing and context management. [4.2]

Dialog level: design level describing temporal aspects and interdependencies in the dialog. [1.2]

- Evaluability:** a measure for how easy it is to evaluate the system. [3.2]
- Evaluation report:** report containing an evaluation of a working system. [5.1]
- Evaluation tool:** tool used to generate an evaluation. [5.1]
- Execution tool:** tool used to bind, interpret, and execute coded modules. [5.1]
- Experimental design:** a design method requiring a running prototype to be evaluated. [3.3]
- External quality:** set of user-perceivable properties of the interface. [1.1]
- FCDE, Functional Core Development Environment:** component supporting the development and testing of the functional core. [1.5]
- FCDE Services:** construction and testing tools in the FCDE. [1.5]
- Flexibility:** the system allows for users' choice during task execution. [2.2]
- Functional Completeness:** all specified tasks are supported by abstract commands and functional state elements, such that the user can solve all specified tasks correctly. [3.2]
- Functional core (FC):** the set of functions performing the application-oriented data processing. [4.2]
- Functional Core Adapter (FCA):** the software component mediating between FC and D. [4.2]
- Functional level:** design level describing operations and objects in a system. [1.2]
- Functional partitioning:** a grouping of the operations for an application domain. [4.2]
- Global SW re-use:** re-use of functional components/modules from other systems. [3.3]
- Goal:** a desired state of the world. [1.2]
- Goal Completeness:** all goals can be reached. [2.2]
- Goal level:** the highest level describing real world goals. [1.2]
- Goal state:** a system state matching the user's goal, which the user tries to achieve (only indirectly defined). [1.2]
- GOMS method:** a design method using a cognitive system model of goals, operators, methods and selection rules. [3.3]
- Honesty:** representations of system state elements are designed to be correctly interpreted. [2.4]
- Human role multiplicity:** the system supports the tasks of multiple human roles simultaneously. [2.3]
- I/O resource management tools:** A program allowing several client programs to share one resource. [3.3]

- I/O Re-use:** the system allows usage of previous I/O as future I/O. [2.3]
- IAS, Interactive system:** A computer system that interacts with one or more human users. [Preface]
- Implementer:** person deciding on over-all implementation, managing and coordinating implementation through the sub-roles: User Interface Implementer and Functional Core Implementer. [1.4]
- Insistence:** system state element representations are preserved until user acknowledgement. [2.4]
- Integration Test:** integration of modules into a final system and testing of the system. [1.3]
- Interaction:** influence of tools and materials on properties and the ease of obtaining specific properties. [5.2]
- Interaction point:** an observable hiatus in an interaction trace. [1.2]
- Interaction trace:** a sequence of concrete steps bringing the system into the goal state. [1.2]
- Inter-application communication standards:** Rules that allow sharing of control and data between applications. [3.3]
- Internal quality:** set of software properties of the interface. [1.1]
- ISDE, Interactive Software Development Environment:** a general and comprehensive environment for the total development. [1.5]
- ISDE services:** a collection of tools supporting development and testing at a high level. [1.5]
- Iterative design:** a design method where each development version is evaluated by users. [3.3]
- Logical Interaction (LI):** the software component mediating between D and PI. [4.2]
- Logical interaction level:** design level describing the dialog in presentation entities. [1.2]
- Maintainability:** a measure for how easy it is to manage and maintain the finished system. [3.2]
- Materials:** All kinds of documents and code produced for a specific project. [5.1]
- Migratability:** the initiative for abstract command execution can be transferred between user and system. [2.3]
- Modifiability:** a measure for how easy it is to modify a system, i.e. change its functionality. [3.2]
- Module design:** refinement of the system model into software modules. [1.3]
- Module test:** testing that modules meet specifications. [1.3]

- Multi-threading:** the user can engage in several tasks simultaneously. [2.3]
- Non-preemptiveness:** the user has a choice of the next interaction step within current task execution. [2.3]
- Observability:** all relevant system state elements are perceivable by the users. [2.4]
- Observable state:** the observable part of the system state. [1.2]
- Pace Tolerance:** the user may control the pace of interaction. [2.4]
- Physical Interaction (PI):** the software component implementing the physical interaction between user and system. [4.2]
- Physical interaction level:** the lowest design level describing all the physical dialog events in detail. [1.2]
- Portability:** a measure for how easy it is to change hardware or software environment of the system. [3.2]
- Predictability:** the user can predict future states and response times from the current perceivable state. [2.4]
- Predictive design:** a design method where early versions of the design are inspected by specialists. [3.3]
- Problem analysis:** identification of the problem to be solved. [1.3]
- Project manager:** person providing resources for the project. [1.4]
- Property addressing:** developer effort (requiring human factor skills) to construct a system such that it possesses the property. [5.2]
- Property assessment:** work carried out by a system developer to assess that a system has the property. [4.1]
- Property assistance:** an architecture assists a property, if the developer with some effort can build a system having the property. [4.1]
- Property proof:** developer effort (requiring skills in formalizing properties) to verify that a system possesses the property. [5.2]
- Property provision (or property delivery):** an architecture provides a property, if the system built in that architecture possesses the property without further developer effort. [4.1]
- Quality assurance:** method or procedure for testing quality of interactive systems. [3.3]
- Reachability:** users can reach any state from any other state.
- Reconfigurability:** the user can initiate customization of the interaction. [2.3]
- Rendering:** a sequence of system actions to present an observable state. [1.2]

- Representation Multiplicity:** the system offers alternative representations of I/O objects. [2.3]
- Requirements (materials):** documents specifying the requirements of an interactive system. [5.1]
- Requirements specialist:** person performing needs and task analysis and transforming the users' conceptual models into system requirements. [1.4]
- Requirements specification:** capture of constraints and requirements for the intended system. [1.3]
- Requirement tool:** tool that may assist in capturing requirements. [1.5]
- Robustness:** the system facilitates users' actions and helps the user out of mistakes. [2.2]
- Run time Efficiency:** a measure for how efficient the system uses the computer resources. [3.2]
- Seeheim:** architectural metamodel for interactive systems. [4.4]
- Software design/Global SW design:** transformation of a system model into a global software structure. [1.3]
- Specification language:** formal language for specifying interfaces. [3.3]
- Specifications and design (materials):** documents with a detailed specification of an IAS. [5.1]
- Specification tool:** tool used to transform requirements into specifications. [5.1]
- System acceptance:** monitoring and helping the users to use the system. [1.3]
- System administrator:** person responsible for keeping the system running and providing maintenance. [1.4]
- System design:** transformation of requirements into a solution expressed as a system model. [1.3]
- System designer:** person making the initial system level design and coordinating the sub-roles: User Interface Designer and Functional Core Designer. [1.4]
- System state:** the internal state of a computer system. [1.2]
- System test:** checking that the system meets the external specifications. [1.3]
- Target environment:** the HW/SW platform on which the interactive system is finally installed. [3.3]
- Task:** a concrete activity that can lead to a goal state. [1.2]
- Task Completeness:** all goals identified in specified scenarios are attainable with known task methods. [2.2]

- Task level:** design level describing tasks by means of which one can achieve the goals (only indirectly defined). [1.2]
- Task support:** any feature or action by a person that supports task execution. [1.2]
- Tools:** Artefacts used by developers to produce materials. [5.1]
- UIDE, User Interface Development Environment:** component supporting the development and testing of the user interface part. [1.5]
- UIDE Services:** construction and testing tools in the UIDE. [1.5]
- UIMS:** user interface management system. [1.5]
- UIS, User Interface System:** that part of an interactive system that manages the dialog and performs the dialog functions. [1.5]
- User:** end-user of the final system. [1.4]
- User interface integratability:** a measure for how easy it is to integrate the system with other application systems. [3.2]
- User interface standard:** a rule – more or less widely accepted and formalized – that encapsulates good engineering practice. [3.3]
- User representative:** a person with domain knowledge participating in design and usability testing. [1.4]
- V-model:** a phase model for software development. [1.3]
- V-model with backtracking:** an iterative V-model with recovery steps. [1.3]
- Validator:** person responsible for the quality plan and its implementation. May be split into the sub-roles: Quality Specialist who plans and manages the testing; Usability Specialist who plans and conducts usability evaluation and user testing; Software Validator who plans and manages testing of interface software. [1.4]
- Working system:** the set of coded modules when linked together into a running IAS. [5.1]

APPENDIX B

Summary Tables

Table 2.1 *Summary of Flexibility Properties.*

Flexibility Property	Description	Related properties
Representation:		
Device multiplicity	More than one way to do something	Multi-media capability
Representation multiplicity	More than one way to present something	I/O multiplicity, equal opportunity, multi-modality
Input/Output re-use	History repeating itself	Use of defaults
Planning:		
Human role multiplicity	Several people doing several things	Access control
Multithreading	One person doing several things	Concurrency, interleaving
Non-preemptiveness	Doing what you want when you want	User-driven dialog, mixed-initiative dialog
Reachability	Getting anywhere from anywhere else	Commensurate effort
Adaptivity:		
Reconfigurability	The user changing the interaction	Programmability of the interface
Adaptivity	The system changing the interaction	Automatic macro construction
Migratability	Transferring control	

Table 2.2 *Summary of Robustness Properties.*

Robustness Property	User dep.	Description	Related properties
Observability	+	The user may perceive	Immediacy, browsability, feedback, feedthrough
Insistence	+	The user will perceive	Salience, timeliness, persistence, awareness
Honesty	++	The user correctly comprehends	Affordance, familiarity, suggestiveness, guessability
Predictability	+	Understanding how the system will react	Observability, consistency, affordance, response time stability
Access control		Role-sensitive restriction of information availability	Human role multiplicity, feedthrough, awareness, visibility, privacy
Pace tolerance		Response times match user's expectations	Timeliness, adaptivity, migratability
Deviation tolerance		User's recovery intentions are supported	Forward/backward recoverability, commensurate effort, pre-emptiveness

Table 3.1 Relationships between Internal Properties and Software Techniques.

Internal Property	The Use of Software Techniques								
	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		+	+	-	++				

Table 3.2 Relationships between Flexibility Properties and Software Techniques.

Flexibility Property	The Use of Software Techniques								
	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	++	++			-			+	+

Table 3.3 Relationships between Robustness Properties and Software Techniques.

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	+	++	+		++		-	+	

Table 5.1 *Specification Interaction between Tools/Materials and Flexibility.*

Property	Interaction	Comment
Reachability	Prove	Most straightforward with 'clean' dialog abstractions
Non-preemptiveness	Assess	By inspecting specifications that support proofs of reachability
	Deliver	By using dialog abstractions with process constructs
Multi-threading	Obstruct	By using any sequential dialog abstraction
	Address	By using dialog abstractions with process constructs
Device Multiplicity, I/O Re-use and Human Role Multiplicity	None	Dependent on construction tools/materials
Representation Multiplicity	Assess	Same relationships as observability with constraints, view controllers, model-based tools and cognitive walkthrough
Reconfigurability, Adaptivity and Migratability	None	Dependent on construction tools/materials

Table 5.2 *Specification Interactions between Tools/Materials and Robustness.*

Property	Interaction	Comment
Observability	Address Assess	Constraints/View Controllers Model-Based User Interface Generators Cognitive Walkthrough Questions 2 and 4
Insistence	Assess	Cognitive Walkthrough Questions 2 and 4
Honesty	Assess	Cognitive Walkthrough Question 4 Temporal aspects assessed in conjunction with both observability and response time conformance
Predictability	Assess	Cognitive Complexity Theory (but effectiveness disputed (Knowles, 1988)) Response Time Stability assessed along with pace tolerance Cognitive Walkthrough Question 3
Access Control	None	Dependent on construction materials
Pace Tolerance	Deliver	Real time scheduling algorithms (potentially)
Deviation Tolerance	Address Address Assess	Partial support from UI management tools/builders with input validation construct Pre-conditions as used in NUF (Cockton <i>et al.</i> , 1995) and Model-Based User Interface Generators Cognitive Walkthrough can establish effects of errors

Table 5.3 *Specification Interaction between Tools/Materials and Internal Properties.*

Property	Interaction	Comment
Development Efficiency	Deliver	UI Management Tools/Builders with validated efficiency, but such tools are rare Model-based UI generators (mostly research and industrial prototypes) Appropriate specification abstractions, but only dialog level abstractions are well established Detailed unambiguous style guides, but these are rare (toolkit implementations for construction are better)!
System Modifiability	Deliver	Architectural refinement, but only for anticipated potential changes Model-based UI generators (mostly research and industrial prototypes) Hypertext requirements linking tools such as RETH (Kaindl, 1993)
User Interface Integratability	Deliver	Limited support from general (UI) tools
Run Time Efficiency	Deliver	Virtual separation
Portability, Evaluability and Maintainability	None	Preservation of property from architectural model

Table 5.4 *Construction Interaction between Tools/Materials and Flexibility.*

Property	Interaction	Comment
Device Multiplicity	Deliver	Resource Manager, but often restricted to specific drivers in window systems, unless Plug and Play supported
Representation Multiplicity	Assist	By View Controllers (SERPENT), but mostly support from materials (e.g. Model-View Controller (Smalltalk), Multi-View Agents)
I/O Re-use	Assist	Inter-Application communication facilities, if compatibility problems avoided Object Linking and Embedding
Human-Role Multiplicity	Assist	By groupware toolkits
Multi-threading	Deliver	Resource Manager
Non-preemptiveness	Assist	Resource Manager, but pre-emptiveness can be obstructed
Reachability	Deliver	By re-usable history module (or class)
Reconfigurability	Obstruct	By virtual toolkits, but situation is improving
	Assist	By table-driven software, macro recording, feature modification (e.g. changing menu items) and tools such as Tcl/Tk
Adaptivity and Migratability	Assist	Limited support from materials (e.g. User Modeling Shells, Plug and Play, AgentWare?)

Table 5.5 *Construction Interaction between Tools/Materials and Robustness.*

Property	Interaction	Comment
Observability	Assist	Generally, T/Ms supporting representation multiplicity (e.g. view controllers) support observability Also assisted by context-sensitive help and UIMS with Arch/Slinky architecture
Insistence	Deliver	By very specialized materials (e.g. materials for modal dialog boxes or repeated audio replay)
Honesty	Assist	Generally, T/Ms supporting representation multiplicity, response-time stability and pace tolerance support honesty
Predictability	Deliver	Percent-done code delivers partial and very specialized support (response-time conformance, also achievable by reducing resource usage)
Access Control	Deliver	By access control lists
	Assist	By customized overlays as well as by more basic file system features
Pace Tolerance	Deliver	Delay introducing operations (e.g. for reading messages)
Deviation Tolerance	Assist	By 'clean' dialog abstractions that support processes, by constructs for error recovery such as fail-safe programming language features
	Obstruct	By resource managers that silently ignore errors in configuration files (e.g. X Window System)

Table 5.6 *Construction Interactions between Tools/Materials and Internal Properties.*

Property	Interaction	Comment
Development Efficiency	Deliver	Well-designed tools and materials should always deliver this property
System Modifiability	Assess	By inspection techniques, but largely an architectural property
Portability	Assist	By virtual toolkits and more generally by layered wrappers or emulations and simulators
Evaluability	Deliver	Instrumentation code
Maintainability	Deliver	Instrumentation code reveals common problems
	Assist	By Inspection Techniques
Run Time Efficiency	Obstruct	By Instrumentation code, layered wrappers and emulations/simulators, which slow things down
	Assist	By virtual separation, which removes layers at run time
User Interface Integratability	Assist	By standardized (style-guide-based) components and other common components By tools such as Visual Basic (Microsoft) and Tcl/Tk By materials such as inter-application communication facilities and Object Linking and Embedding (Microsoft)

References

- Ahlberg, C. and Schneiderman, B. (1994). Visual information seeking: Tight coupling of dynamic query filters with starfield displays. In Adelson, B., Dumais, S., and Olson, J., editors, *Proceedings of CHI'94 Human Factors in Computing Systems*, pages 313–17.
- Alty, J. (1991). Multimedia: What is it and how do we exploit it? In Diaper, D. and Hammond, N., editors, *People and Computers VI, Proceedings of the HCI'91 conference*, pages 31–44. Cambridge University Press.
- Alty, J. L. (1984). The application of path algebras to interactive dialogue design. *Behaviour and Information Technology*, 3(2): 119–32.
- Alty, J. L. and Ritchie, R. A. (1985). A path algebra support facility for interactive dialogue designers. In Cook, S. and Johnson, P., editors, *People and Computers: Designing the Interface*, pages 128–37. Cambridge University Press, Cambridge.
- Apple (1991). *Macintosh User's Guide for Desktop Computers*. Apple Computer Inc., Ireland. Document Z030-1751-A.
- Apple (1992). *Human Interface Guidelines: The Apple Desktop Interface*. Addison-Wesley.
- Apple (1993). *Inside Macintosh: Interapplication Communication*. Addison-Wesley.
- Bannon, L. and Bødker, S. (1991). Beyond the interface: Encountering artifacts in use. In Carroll, J., editor, *Designing Interaction*, pages 227–53. Cambridge University Press, Cambridge.
- Bass, L. and Coutaz, J. (1991). *Developing Software for the User Interface*. Addison-Wesley.
- Bass, L., Clapper, B., Hardy, E., Kazman, R. and Seacord, R. (1990). Serpent: A user interface management system. In *Proceedings of Winter 1990 USENIX Conference*, pages 245–57. USENIX Association, Berkeley.
- Beech, D., editor (1986). *Concepts in User Interfaces: A Reference Model for Command and Response Languages*, volume 234 of *Lect. Notes in Comp. Sc.* Springer-Verlag, New York.

- Benyon, D. (1992). The role of task analysis in systems design. *Interacting with Computers*, 4(1): 102-23.
- Berlage, T. and Spenke, M. (1992). The GINA interaction recorder. In Larson, J. and Unger, C., editors, *Engineering for Human-Computer Interaction '92*, pages 69-80. IFIP Transactions A-18. North-Holland, Amsterdam.
- Bernsen, N. O. (1993). Modality theory: supporting multimodal interface design. In *Workshop ERCIM on Multimodal Human-Computer Interaction, INRIA, Lorraine*. Chambery.
- Bersoff, E. H. and Davis, A. M. (1991). Impacts of life cycle models on software configuration management. *Communications of ACM*, 34(8): 104-17.
- Bevan, N. (1983). The design of user-friendly systems for generating intelligent dialogues in integrated interactive computing systems. In Degano, P. and Sandewall, E., editors, *Proceedings of ECICS'82*, pages 333-44. North-Holland, Amsterdam.
- Blattner, M. M. and Dannenberg, R. B., editors (1992). *Multimedia Interface Design*. ACM Press Frontier Series. Addison-Wesley.
- Boehm, B. W. (1988). A spiral model of software development and enhancement. *COMPUTER*, 21(5): 61-72.
- Burns, A. (1994). Preemptive priority based scheduling: An appropriate engineering approach. In Son, S., editor, *Advances in Real-Time Systems*, pages 225-48. Prentice Hall.
- Card, S. K., Moran, T. P., and Newell, A. (1983). *The Psychology of Human-Computer Interaction*. Lawrence Erlbaum Associates.
- Carroll, J. M. and Carrithers, C. (1984). Training wheels in a user interface. *Communications of ACM*, 27(8): 800-6.
- Carroll, J. M. and Rosson, M. B. (1991). Deliberated evolution: stalking the view matcher in design space. *Human-Computer Interaction*, 6(3 & 4): 281-318.
- Chen, M. (1993). A framework for describing interactions with graphical widgets. In Ashlund, S., Mullet, K., Henderson, A., Hollnagel, E. and White, T., editors, *Human Factors in Computing Systems. Proceedings of INTERCHI'93*, pages 131-2. ACM Press, New York.
- Clarke, S., Jordan, P. and Cockton, G. (1995). Applying Aristotle's theory of poetics to design. In Lovesey, E., editor, *Proceedings of UK Ergonomics Conference: Contemporary Ergonomics*, pages 139-44. Taylor & Francis.

- Cockton, G. (1985). Three transition network dialogue management systems. In Johnson, P. and Cook, S., editors, *People and Computers: Designing the interface*, pages 135–44. Cambridge University Press, Cambridge.
- Cockton, G. (1987a). Interaction ergonomics, control and separation: Open problems in user interface management. *Information and Software Technology*, 29(4): 176–91.
- Cockton, G. (1987b). A new model for separable interactive systems. In Bullinger, H.-J. and Shackel, B., editors, *Human-Computer Interaction - INTERACT'87*, pages 1033–38 (participants edition). North-Holland, Amsterdam.
- Cockton, G. (1991). The architectural bases of design re-use. In Duce, D., Gomes, M., Hopgood, F. and Lee, J., editors, *User Interface Management and Design*, pages 15–34. Springer-Verlag, Berlin.
- Cockton, G., Clarke, S., Gray, P. and Johnson, C. (1995). Literate development: Weaving human context into design specifications. In Benyon, D. and Palanque, P., editors, *Critical Issues in User Interface Systems Engineering*. Springer-Verlag, Berlin.
- Coutaz, J. (1987). PAC, an implementation model for dialogue design. In Bullinger, H.-J. and Shackel, B., editors, *Human-Computer Interaction - INTERACT'87*, pages 431–6 (participants edition). North-Holland, Amsterdam.
- Coutaz, J., Duce, D., Duke, D., Faconti, G., Harrison, M., Nigay, L., Paterno, F. and Salber, D. (1995). The Amodeus System Reference Model. Technical report, Amodeus Project. Amodeus Project Document: System Modelling/WP54.
- Cypher, A. (1991). Eager: Programming repetitive tasks by example. In *Human Factors in Computing Systems, CHI'91 Conference Proceedings*, pages 33–9. ACM Press.
- Dewan, P. (1993). Tools for implementing multiuser user interfaces. In Bass, L. and Dewan, P., editors, *Trends in Software: Issue on User Interface Software*, 1: 149–72.
- Dewan, P. and Choudhary, R. (1995). A general multi-user undo/redo model. In Marmolin, H., Sundblad, Y. and Schmidt, K., editors, *Proceedings of European Conference on Computer Supported Work*, pages 231–46. Kluwer, Dordrecht.
- Dewan, P. and Shen, H. (1992). Access control for collaborative environments. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work*, pages 51–8. ACM Press.

- Diaper, D., editor (1989). *Task Analysis for Human-Computer Interaction*. Ellis Horwood.
- Dix, A. (1991). *Formal Methods for Interactive Systems*. Academic Press.
- Dix, A. (1994). CSCW – a framework. In Rosenburg, D. and Hutchinson, C., editors, *Design Issues in CSCW*. Springer-Verlag.
- Dix, A., Finlay, J., Abowd, G. and Beale, R. (1993). *Human-Computer Interaction*. Prentice Hall International.
- Dougherty, D., Koman, R. and Ferguson, P. (1994). *The Mosaic Handbook for the X Window System*. O'Reilly and Associates.
- England, D. (1988). Graphical prototyping of graphical tools. In Jones, D. and Winder, R., editors, *People and Computers IV*, pages 407–20. Cambridge University Press, Cambridge.
- Foley, J., Wallace, V. and Chan, P. (1987). The human factors of computer graphics interaction techniques. *IEEE Computer Graphics and Applications*, 4(11): 13–48.
- Fraser, C. W. and Krishnamurthy, B. (1990). Live text. *Software – Practice and Experience*, 20(8): 851–8.
- Garlan, D. and Shaw, M. (1993). An introduction to software architecture. *Advances in Software Engineering and Knowledge Engineering*, 1.
- Gaver, W. and Smith, R. (1990). Auditory icons in large-scale collaborative environments. In Diaper, D., Gilmore, D., Cockton, G. and Shackel, B., editors, *Proceedings of INTERACT'90*, pages 735–40. Elsevier Science Publishers.
- Gibbs, S. J. (1989). LIZA: An extensible groupware toolkit. In *Proceedings of CHI'89*, pages 29–35. ACM Press.
- Goodman, D. (1993). *The complete Hypercard handbook*. Random House, New York.
- Gosling, J. and McGilton, H. (1995). *The Java Language Environment: A white paper*. Sun Microsystems Inc.
- Gourdol, A., Nigay, L., Salber, D. and Coutaz, J. (1992). Two case studies of software architecture for multimodal interactive systems: VoicePaint and voice-enabled Graphical Notebook. In Larson, J. and Unger, C., editors, *Engineering for Human-Computer Interaction '92*, pages 271–84. IFIP Transactions A-18. North-Holland, Amsterdam.
- Gram, C. (1995). A software engineering view of user interface design. In Bass, L. and Unger, C., editors, *Engineering for Human-Computer*

- Interaction '95*, pages 293-306. IFIP Working Group 2.7, Chapman & Hall.
- Green, M. (1985). Design notations and user interface management systems. In Pfaff, G. E., editor, *User Interface Management Systems*, pages 89-107. Springer-Verlag.
- Harel, D. (1988). On visual formalismss. *CACM*, 31(5): 514-30.
- Hartson, H. R. and Gray, P. D. (1992). Temporal aspects of tasks in the user action notation. *Human-Computer Interaction*, 7: 1-45.
- Hill, R. D. (1987). Event-response systems - a technique for specifying multi-threaded dialogues. In *Human Factors and Computing Systems - Proceedings of CHI+GI'87*, pages 241-8. ACM Press.
- Hill, R. (1992). The Abstraction-Link-View paradigm: Using constraints to connect user interfaces to applications. In *Proceedings of CHI '92*, pages 335-42. ACM Press.
- Hix, D. and Hartson, H. R. (1994). IDEAL: An environment to support usability in human-computer interaction. In Blumenthal, B., Gornostaev, J. and Unger, C., editors, *Proceedings of EWHCI'94*, pages 95-106. Springer-Verlag.
- ISO (1987). *ISO9000, Quality Control*. ISO, International Standards Organization. A series of standards on quality assurance and requirements to quality control systems.
- Jacob, R. J. K. (1986). A specification language for direct-manipulation user interfaces. *ACM Transactions on Graphics*, 5(4): 283-317.
- Johnson, P., Johnson, H. and Wilson, S. (1995). Rapid prototyping of user interfaces driven by task models. In Carroll, J., editor, *Scenario-Based Design*, pages 209-46. John Wiley, New York.
- Jordan, P. W., Draper, S. W. and MacFarlane, K. K. (1991). Guessability, learnability and experienced user performance. In Diaper, D. and Hammond, N., editors, *People and Computers VI, Proceedings of HCI'91*, pages 237-48, Cambridge University Press, Cambridge.
- Kaindl, H. (1993). The missing link in requirements engineering. In *Software Engineering Notes*, pages 30-9. ACM Press, New York.
- Kazman, R., Bass, L., Abowd, G. and Webb, M. (1994). SAAM: A method for analyzing the properties of user interface software architectures. In *Proceedings of the 16th International Conference on Software Engineering*, pages 81-90, Sorrento, Italy.
- Kim, W. C. and Foley, J. D. (1993). Providing high-level control and expert assistance in the user interface presentation design. In Ashlund,

- S., Mullet, K., Henderson, A., Hollnagel, E. and White, T., editors, *Human Factors in Computing Systems, Proceedings of INTERCHI'93*, pages 424–9. ACM Press, New York.
- Knister, M. J. and Prakash, A. (1990). Distedit: A distributed toolkit for supporting multiple group editors. In *Proceedings of ACM CSCW'90 Conf. on Computer-Supported Cooperative Work*, System Infrastructure for CSCW, page 343. ACM Press, New York.
- Knowles, C. (1988). Can cognitive complexity theory (CCT) produce an adequate measure of system usability? In Jones, D. M. and Winder, R., editors, *People and Computers IV*, pages 291–307. Cambridge University Press, Cambridge.
- Kobsa, A. (1990). Modeling the user's conceptual knowledge in bgp-ms, a user modelling shell system. *Computational Intelligence*, 6.
- Krasner, G. E. and Pope, S. T. (1988). A cookbook for using Model-View-Controller user interface paradigm in Smalltalk-80. *Journal of Object Oriented Programming*, August/September: 26–49.
- Krell, E. and Krishnamurthy, B. (1992). COLA: Customized Overlaying. In *USENIX San Francisco Winter 1992 Conference Proceedings*, pages 3–7. USENIX Association, Berkeley.
- Kühme, T. and Schneider-Hufschmidt, M. (1992). SX/Tools – an open design environment for adaptable multimedia user interfaces. *Computer Graphics Forum*, II(3). EUROGRAPHICS'92 Conference Issue.
- Kühme, T., Dieterich, H., Malinowski, U. and Schneider-Hufschmidt, M. (1992). Approaches to adaptivity in user interface technology. In Larson, J. and Unger, C., editors, *Engineering for Human-Computer Interaction '92*, pages 225–52. IFIP Transactions A-18. North-Holland, Amsterdam.
- Lantz, K., Tanner, P., Binding, C., Huang, K. and Dwelly, A. (1987). Window systems, reference models and concurrency. *Computer Graphics*, 21(2): 87–97.
- Lauwers, J. C. and Lantz, K. A. (1990). Collaboration awareness in support of collaboration transparency: Requirements for the next generation of shared window systems. In *Proceedings of ACM CHI'90*, pages 303–12. ACM Press, New York.
- Long, J. (1989). Cognitive Ergonomics and Human-Computer Interaction. In Warr, P., editor, *Psychology at Work*. Penguin, Harmondsworth.
- Luckham, D., Kenney, J., Augustin, L., Vera, J., Bryan, D. and Mann, W. (1995). Specification and analysis of system architecture using rapide. *IEEE Transaction on Software Engineering*.

- Macromind (ca. 1990). *MacroMind Director*. Macromind Inc., San Francisco.
- Meyer, B. (1988). *Object-oriented Software Construction*. Prentice-Hall, Hemel Hempstead.
- Microsoft (1992). *The Windows Interface: An Application Design Guide*. Microsoft Corporation.
- Microsoft (1993a). *Windows Application Design Guide*. Microsoft Corporation.
- Microsoft (1993b). *Visual Basic, Programmer's Guide*. Microsoft Corporation.
- Microsoft (1995). *The Windows Interface Guidelines for Software Design*. Microsoft Corporation.
- Monk, A., Wright, P., Haber, J. and Davenport, L. (1993). *Improving your Human-Computer Interface*. Prentice Hall.
- Moran, T. (1981). The command language grammar: a representation for the user interface of interactive systems. *International Journal of Man-Machine Studies*, 15(1): 3–50.
- Muller, M. J., Wildman, D. M. and White, E. A. (1993). Taxonomy of PD Practices: A Brief Practitioner's Guide. *Communications of ACM*, 36(4): 26–8.
- Myers, B. A. (1985). The importance of percent-done progress indicators for computer-human interfaces. In *CHI '85, Proceedings of ACM SIGCHI Conference*, pages 11–17. ACM Press, New York.
- Myers, B. A. (1988). *Creating User Interfaces by Demonstration*. Academic Press, Boston.
- Newman, I. A. and Smith, P. A. (1995). Meeting and managing user expectations in large scale distributed systems: a case study. Technical report, Midlands Regional Research Laboratory, Loughborough University of Technology. Genie Research Report 3.
- Newman, I. A., Campbell, S. P., Medyckyj-Scott, D. J., Parks, L. M. and Smith, P. A. (1995). Building large scale federal distributed information management systems – a method (TIMES) and an application (GENIE). Technical report, Midlands Regional Research Laboratory, Loughborough University of Technology. Genie Research Report 2.
- Nielsen, J. (1992). Finding usability problems through heuristic evaluation. In *Proceedings of CHI'92*, pages 373–80. ACM Press, New York.
- Nielsen, J. (1993). *Usability Engineering*. AP Professional, Boston.

- Nielsen, J. (1994). Enhancing the explanatory power of usability heuristics. In *Proceedings of CHI'94*, pages 152–8. ACM Press, New York.
- Nigay, L. (1994). *Conception et modélisation logicielles des systèmes interactifs: application aux interfaces multimodales*. PhD thesis, University of Grenoble.
- Nigay, L. and Coutaz, J. (1993). A design space for multimodal systems: Concurrent processing and data fusion. In *Human Factors in Computing Systems, Proceedings of INTERCHI'93*, pages 172–8. ACM, Addison-Wesley.
- Nigay, L. and Coutaz, J. (1995). A generic platform for addressing the multimodal challenge. In *Proceedings of CHI'95*, pages 98–105. ACM/SIGCHI, ACM Press.
- Norman, D. A. (1986). Cognitive engineering. In Norman, D. A. and Draper, S. W., editors, *User-Centred Systems Design*. Erlbaum Associates, Hillsdale, NJ.
- Norman, D. (1988). *The Psychology of Everyday Things*. Basic Books.
- OSF (1990). *OSF/Motif Style Guide*. Prentice-Hall. (One of five books on the Motif user interface.)
- Ousterhout, J. K. (1994). *Tcl and the Tk Toolkit*. Addison-Wesley.
- Pfaff, G., editor (1985). *User Interface Management Systems*. Springer-Verlag, New York.
- Pfaffenberger, B. (1995). *Netscape Navigator*. AP Professional, Boston. (Also available on CD-ROM. Netscape documentation is found on World Wide Web on URL-address <http://merchant.netscape.com/netstore/pubs/index.html>.)
- Polson, P., Lewis, C., Rieman, J. and Wharton, C. (1992). Cognitive walk-throughs: A method for theory-based evaluation of user interfaces. *Internat. Journal of Man-Machine Studies*, 36(5): 741–73.
- Reiss, S. P. (1990). Connecting tools using message passing in the field environment. *IEEE Software*, 7(4): 57–66.
- Retter, P., Mousel, P. and Nogacki, G. (1992). Maximum abstraction as a path towards portability in multiple graphical environments. In Larson, J. and Unger, C., editors, *Engineering for Human-Computer Interaction '92*, pages 51–68. IFIP Transactions A-18. North-Holland, Amsterdam.
- Sadler, H. J. (1993). Making it Macintosh: An interactive human interface instructional product for software developers. In Ashlund, S., Mullet, K., Henderson, A., Hollnagel, E. and White, T., editors, *Human*

- Factors in Computing Systems. Proceedings of INTERCHI'93*, pages 37–8. ACM Press, New York.
- Scheifler, R. W., Gettys, J., Flowers, J. and Rosenthal, D. (1992). *X Window System*. Digital Press, 3rd edition.
- Schmucker, K. (1986). MacApp: an application framework. *Byte*, 11(8): 189–93.
- Sha, L. and Sathaye, S. (1993). Distributed real-time system design: Theoretical concepts and applications. Technical Report CMU/SEI-93-TR-2 ADA2265199, CMU.
- Shevlin, F. and Neelamkavil, F. (1991). Designing the next generation of UIMSs. In Duce, D., Gomes, M., Hopgood, F. and Lee, J., editors, *User Interface Management and Design*, pages 123–34. Springer-Verlag, New York.
- Siemens Nixdorf (1994). *Dialog Builder User Guide Manual*. Siemens Nixdorf Informationsysteme AG, 81730 Munich, Germany. Manual U20644-J-Z357-2-7600.
- SmethersBarnes (1990). *Prototyper: Reference Manual*. SmethersBarnes, Portland, Oregon. (Developed by G. R. Cossey.)
- Smith, P. A. and Newman, I. A. (1995). Rapid applications implementation and design: the concept, a realisation and some issues. Technical report, Midlands Regional Research Laboratory, Loughborough University of Technology. (Available from the authors.)
- Smith, P. A. and Parks, L. M. (1995). Exemplar user interfaces for the TIMES distributed system builder. Technical report, Midlands Regional Research Laboratory, Loughborough University of Technology. Genie Research Report 6.
- Sukaviraya, P., Isaacs, E. and Bharat, K. (1992). Multimedia help: A prototype and an experiment. In Bauersfeld, P., Bennett, J. and Lynch, G., editors, *Human Factors in Computing Systems: CHI'92 'Striking a Balance'*, pages 433–4. ACM Press, New York.
- Sukaviraya, P., Foley, J. D. and Griffith, T. (1993). A second generation user interface design environment: The model and the runtime architecture. In K. Mullet, S. A., Henderson, A., Hollnagel, E. and White, T., editors, *Human Factors in Computing Systems. Proceedings of INTERCHI'93*, pages 375–82. ACM Press, New York.
- Szczur, M. and Sheppard, S. (1993). TAE Plus: Transportable applications environment plus: A user interface development environment. *ACM Transactions on Information Systems*, 11(1).

- Szekely, P., Luo, P. and Neches, R. (1993). Beyond interface builders: model-based interface tools. In Ashlund, S., Mullet, K., Henderson, A., Hollnagel, E. and White, T., editors, *Human Factors in Computing Systems. Proceedings of INTERCHI'93*, pages 383–90. ACM Press, New York.
- Taylor, R. and Johnson, G. (1993). Separation of concerns in the Chiron-1 user interface development and management system. In *Proceedings of INTERCHI'93*, pages 367–74. ACM Press, New York.
- Thimbleby, H. (1990). *User Interface Design*. Addison Wesley.
- Took, R. K. (1990). *Surface Interaction: Separating Direct Manipulation Interfaces from their Applications*. PhD thesis, Univ. of York.
- Trefz, B. and Ziegler, J. (1989). The user interface management system diamant. In Cockton, G., editor, *Engineering for Human-Computer Interaction '89*, pages 177–96. North-Holland, Amsterdam.
- UIMS (1992). The UIMS tool developers workshop: A metamodel for the runtime architecture of an interactive system. *SIGCHI Bulletin*, 24(1): 32–7.
- Vanderdonckt, J. and Bodart, F. (1993). Encapsulating knowledge for intelligent automatic interaction objects selection. In Ashlund, S., Mullet, K., Henderson, A., Hollnagel, E. and White, T., editors, *Human Factors in Computing Systems. Proceedings of INTERCHI'93*, pages 430–7. ACM Press, New York.
- Wasserman, A. I. (1985). Extending state transition diagrams for the specification of human-computer interaction. *IEEE Transactions on Software Engineering*, SE-11(8): 699–713.
- Wharton, C., Rieman, J., Lewis, C. and Polson, P. (1994). The cognitive walkthrough: A practitioner's guide. In Nielsen, J. and Mack, R., editors, *Usability Inspection Methods*. Wiley.
- Wiecha, C., Bennett, W., Boies, S., Gould, J. and Greene, S. (1990). ITS: A tool for rapidly developing interactive applications. *ACM Transactions on Information Systems*, 8(3): 204–36.
- Williams, A. S. (1994). The OLE 2.0 object model. In *ACM OOPS Messenger, Addendum to the Proceedings of OOPSLA 1993*, pages 68–70. Published as ACM OOPS Messenger, Addendum to the Proceedings of OOPSLA 1993, volume 5, number 2.
- XVT (1991). *XVT Programmer's Manual*. XVT Software Inc., Boulder, Colorado.
- Yang, Y. (1988). Undo support models. *International Journal of Man-Machine Studies*, 28: 457–81.

Index

- Abstract command, 5
- Abstract data type
 - in Chiron-1, 126
- Abstraction
 - component, PAC, 122
 - level, 4, 46
- Acceptance test, 7
- Access control, 37, 42, 86, 108
 - in ATC, 196
- Active value, 154
- Adaptivity, 28, 35, 82, 105
 - in ATC, 204
- Addressing, 91, 140
- Affordance, 40
- Agent, 121
 - PAC, 121, 123, 207
- Air traffic control, 189
- Architectural model, 68
- Arch model, 95, 115
- Arch/Slinky metamodel, 93, 117, 118
- Articulation, 3
- Assessing, 140
- Assessment of property, 91
- Assistance of property, 91
- ATC, 189
 - support system, 190, 192
- ATCo, 190
- Awareness, 40
 - in ATC, 196

- Backward reachability, 34
- Binding services, 19
- Branching, 117, 118
- Browsability, 38
 - in ATC, 196

- Capability, 92
- Chiron-1 architecture, 126

- CLG, 4
- Client, 14
- Client/server, Chiron-1, 126
- Climate control system
 - in Chiron-1, 129
 - functional partitioning, 94
- Coded module material, 134
- Coding, 7
- Cognitive
 - complexity theory, 144
 - walkthrough, 143
- Cognitive walkthrough, 66
- Command, 5
- Command Language Grammar, 4
- Commensurate effort, 35, 45
- Communication standard,
 - interapplication, 73
- Completeness, xiii, 26
 - functional, 55, 62, 77
 - scenario, 27
- Composition, 115
- Concurrent multithreading, 32
- Consistency, 41
- Construction, 136
 - material, 150
 - tool, 135, 150
- Control component, PAC, 122
- CSCW, 39, 40, 119
- Customizability, 28, 82

- D (Dialog), 94
- Dangerous state, 107
- Delivery, 91, 140
- Designer
 - functional core, 15
 - interactive system, 15
 - system, 15
 - user interface, 15

- Development
 - effective, 210
 - efficiency, 55, 62, 78, 110
 - model, 8
 - participative, 11
 - process, 6
- Deviation tolerance, 37, 44, 87, 106
 - in ATC, 194, 199
 - formalized, 48
- Device multiplicity, 28, 29, 80, 103
 - in ATC, 200
- Dialog, 94
 - component, Arch, 116
 - controller, PAC, 121
 - level, 5, 46
- DialogBuilder, 180
- Dispatcher in Chiron-1, 126
- Division of labor, 115
- Domain-Adapter in Arch, 116

- Encapsulation, 115
- Evaluability, 55, 58, 76, 112
- Evaluation, 136
 - report, 134
 - tool, 135
- Evolutionary prototyping, 10
- Execution tool, 135
- Experimental design method, 67
- External property, xiii, 25, 59, 62, 78, 98
 - in ATC, 193
- External quality, 2

- FC, 93
- FCA, 93
- FCDE, 19
- FCX, 22
- Feedthrough, 39
 - in ATC, 196
- Flexibility, xiii, 26, 27, 78, 99, 109
 - in ATC, 200
 - in construction, 153
 - formalized, 46
 - in specification, 138
 - in TAE+, 169
 - in Visual Basic, 173
- Flight management, 190

- Formal model, 46
- Forward reachability, 34
- Functional
 - completeness, 55, 62, 77, 114
 - core, 22, 93, 121
 - adapter, 93, 121
 - development environment, 19
 - level, 4, 46
 - partitioning, 92, 109
- Functionality of system, 92

- Global software
 - design, 7
 - re-use, 68
- Goal, 3
 - completeness, 26, 99
 - in ATC, 194
 - state, 3
- Goals, operations, methods, selections, 4
- GOMS, 4
 - design method, 67
- Guidelines, GUI, 148
- Gulf of evaluation, 40

- Heuristic evaluation, 67
- Honesty, 37, 40, 86, 107
 - in ATC, 198
- Human role, 14
 - multiplicity, 28, 31, 81, 100
 - in ATC, 201

- Impact on property, 91
- Implementation, 152
- Implementer, 16
- Indirection, 112
- Insistence, 37, 39, 84, 106, 130
 - in ATC, 197
 - formalized, 50
- Inspection, 140
 - principle-based, 66
 - style conformance, 66
- Integratability of user interface, 55, 61, 77, 114, 131
- Integration test, 7

- Interaction
 - flexibility, xiii, 27, 78, 99, 109
 - in ATC, 200
 - point, 3, 48
 - robustness, xiii, 37, 83, 106, 109
 - in ATC, 194, 195
 - between SW properties and phenomena, 210
 - trace, 3
- Interaction component, Arch, 116
- Interactive SW development
 - environment, 17
- Interapplication communication
 - standard, 73
- Interleaved multithreading, 32
- Internal property, xiii, 55, 110
 - in specification, 147
 - in Visual Basic, 176
- Internal quality, 2
- Interoperability, 73
- I/O resource management, 71
- I/O re-use, 28, 30, 80, 104
 - in ATC, 202
 - formalized, 47
- ISDE, 17
 - services, 19
- Iterative design, 66
- ITS, 149

- Keystroke level, 5

- Labeled transition system, 46
- Learnability, 51
- Level
 - of abstraction, 4, 46
 - dialog, 5
 - functional, 4
 - keystroke, 5
 - logical, 5
 - physical, 5
 - session, 5
- LI, 94
- Life cycle, xiii
- Logical
 - interaction, 94
 - level, 5, 46
- Machine model, 46
- Maintainability, 55, 59, 111
- Maintenance, 7
- Material, 133
 - construction, 150
 - requirement, 134
 - specification, 134, 138
- Measurement of property, 91
- Metamodel, Arch/Slinky, 93
- Migratability, 28, 36, 82, 105
 - in ATC, 204
- Migration, 117
- Model
 - Arch, 95
 - formal, 46
 - MVC, 95
 - PAC, 95
 - PAC-Amodeus, 95
 - Seattle, 95
 - Seeheim, 95
- Model-based tool, 136
- Modifiability, 55, 56, 74, 111, 131
- Module
 - construction, 7
 - design, 7
 - test, 7
- Multi-agent, PAC, 122
- Multiplicity
 - device, 28, 103
 - in ATC, 200
 - human role, 28, 31, 81, 100
 - in ATC, 201
 - representation, 28, 103, 130
 - in ATC, 200
- Multithreading, 28, 32, 81, 101
 - in ATC, 202
 - formalized, 47
- Multi-user system, 42
- MVC model, 95

- NASA's Goddard Space Flight Center, 166
- Network latency, 43
- Non-preemptiveness, 28, 33, 81, 101
 - in ATC, 203
 - formalized, 47
- Norman's seven stage model, 143

- Observability, 37, 38, 84, 106
 - in ATC, 195
 - formalized, 48
- Observable state, 4, 46, 49
- Obstruction, 141

- PAC
 - agent, 121, 123, 207
 - model, 95
- PAC-Amodeus, 95, 120
 - applied on ATC, 205
 - Dialog component in ATC, 207
 - Functional core in ATC, 207
 - Interaction component in ATC, 207
 - Presentation component in ATC, 207
- Pace tolerance, 37, 43, 87, 108
 - in ATC, 198
- Participative design, 67
- Participative development, 11
- Partitioning, functional, 92
- Physical
 - interaction, 94
 - level, 5, 46
- PI, 94
- Portability, 55, 57, 76, 112
- Predictability, 37, 41, 86, 107
 - formalized, 49
 - in ATC, 199
- Predictive design method, 66
- Pre-emption, 28, 33
- Presentation component
 - Arch, 116
 - PAC, 121, 122
- Principle-based inspection, 66
- Problem analysis, 6
- Project manager, 14
- Proof, 140
- Property
 - addressing, 91, 140
 - assessment, 91, 140
 - assistance of, 91, 151
 - delivery, 91, 140
 - external, xiii, 2, 25, 59, 62, 78, 98
 - in ATC, 193
 - impact on, 91
 - inspection, 140
 - internal, xiii, 2, 55, 110
 - in construction, 163
 - in specification, 147
 - measurement of, 91
 - obstruction, 141
 - profile, 91
 - proof, 140
- Prototyping, 66, 136
 - evolutionary, 10
 - rapid, 10

- Quality, xiii, 1
 - assurance, 69
 - control, 69
 - external, 2
 - internal, 2
 - plan, 8
 - predictable?, 209
 - specialist, 16

- RAPID, 138
- Rapid prototyping, 10
- Reachability, 28, 34, 82, 102
 - in ATC, 203
 - formalized, 46
- Reconfigurability, 28, 35, 82, 104
 - in ATC, 204
- Reference model, ix
- Regression test, 7
- Rendering, 4
- Replanning simulation, 203
- Representation multiplicity, 28, 29, 80, 103, 130
 - in ATC, 200
- Requirement
 - analysis, 7
 - material, 134
 - specialist, 14, 137
 - specification, 7
 - tool, 135
- Resource management, 71
- Re-use
 - of I/O, 28, 30, 80, 104
 - in ATC, 202
 - of software, global, 68
- Robustness, xiii, 26, 37, 83, 106, 109
 - in ATC, 194, 195

- in construction, 158
 - formalized, 48
 - in specification, 142
 - in TAE+, 169
 - in Visual Basic, 175
- Roles in ATC, 201
- Run time efficiency, 55, 61, 77, 112

- Safe default action, 194
- Safety requirement in ATC, 194
- Scenario completeness, 27
- Seattle model, 95
- Seeheim model, 95, 113
- Separation of concerns, 115
- Serpent architecture, 124
- Session level, 5
- Software
 - design, 7
 - methodology, 65
 - re-use, 68
 - standard, 65
 - technique, 65
 - tool, 65
 - validator, 17
- Specification, 136
 - language, 70
 - material, 134, 138
 - tool, 70, 135, 138
- Standard, 65, 72, 73
- StartView, 184
- State, 4, 46
 - 'dangerous', 107
 - goal, 3
 - machine, 46
 - observable, 4, 49
 - trajectory, 48
 - transition, 46
- Style conformance inspection, 66
- Summative evaluation, 67
- Support of property, 91
- System
 - acceptance, 7
 - administration, 59
 - administrator, 17
 - design, 7
 - designer, 15, 137
 - functionality, 92
 - state, 4, 46
 - test, 7
- TAE+, 166
 - flexibility, 169
 - robustness, 169
- Target environment, 71
- Task, 3
 - completeness, 26, 62, 99
 - description, 3
 - migratability, 36
 - support, 3
- Tcl/Tk, 171, 180
- Temporal stability, 42
- TIMES Distributed System, 180
- Tool, 134
 - construction, 135, 150
 - evaluation, 135
 - execution, 135
 - model-based, 136
 - requirement, 135
 - specification, 135, 138
- Toolbook, 183
- Trace, interaction, 3
- Trajectory of states, 48
- Transition
 - function, 46
 - system, labeled, 46
- Type-ahead, 43

- UIDE, 19
 - services, 21
- UIMS, 18
- UIS, 22
- Usability specialist, 16
- Usage observation, 67
- User, 17
 - representative, 14
 - satisfaction, 51
- User interface
 - design method, 66
 - development environment, 19
 - integratability, 55, 61, 77, 114, 131
 - management system, 18
 - part, 22
 - standard, 72

V-model, 8
 with backtracking, 9
Validator, 16, 17
Virtual separation, 147
Visual Basic, 172
 flexibility, 173
 internal property, 176
 at research centre, 183
 robustness, 175

Waterfall model, 8
What-if scenario, 203
Working system material, 134