



Discovering Requirements

How to Specify
Products and
Services

Ian Alexander
Ljerka Beus-Dukic



Discovering Requirements

***How to Specify
Products
and Services***

Ian Alexander
and
Ljerka Beus-Dukic



A John Wiley and Sons, Ltd., Publication

Discovering Requirements

Discovering Requirements

***How to Specify
Products
and Services***

Ian Alexander
and
Ljerka Beus-Dukic



A John Wiley and Sons, Ltd., Publication

Copyright © 2009

John Wiley & Sons Ltd, The Atrium, Southern Gate, Chichester,
West Sussex PO19 8SQ, England
Telephone (+44) 1243 779777

Email (for orders and customer service enquiries): cs-books@wiley.co.uk
Visit our Home Page on www.wileyeurope.com or www.wiley.com

All Rights Reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except under the terms of the Copyright, Designs and Patents Act 1988 or under the terms of a licence issued by the Copyright Licensing Agency Ltd, 90 Tottenham Court Road, London W1T 4LP, UK, without the permission in writing of the Publisher. Requests to the Publisher should be addressed to the Permissions Department, John Wiley & Sons Ltd, The Atrium, Southern Gate, Chichester, West Sussex PO19 8SQ, England, or emailed to permreq@wiley.co.uk, or faxed to (+44) 1243 770620.

Designations used by companies to distinguish their products are often claimed as trademarks. All brand names and product names used in this book are trade names, service marks, trademarks or registered trademarks of their respective owners. The Publisher is not associated with any product or vendor mentioned in this book.

This publication is designed to provide accurate and authoritative information in regard to the subject matter covered. It is sold on the understanding that the Publisher is not engaged in rendering professional services. If professional advice or other expert assistance is required, the services of a competent professional should be sought.

Other Wiley Editorial Offices

John Wiley & Sons Inc., 111 River Street, Hoboken, NJ 07030, USA

Jossey-Bass, 989 Market Street, San Francisco, CA 94103-1741, USA

Wiley-VCH Verlag GmbH, Boschstr. 12, D-69469 Weinheim, Germany

John Wiley & Sons Australia Ltd, 42 McDougall Street, Milton, Queensland 4064, Australia

John Wiley & Sons (Asia) Pte Ltd, 2 Clementi Loop #02-01, Jin Xing Distripark, Singapore 129809

John Wiley & Sons Canada Ltd, 6045 Freemont Blvd, Mississauga, ONT, L5R 4J3, Canada

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic books.

Library of Congress Cataloging-in-Publication Data:

Alexander, Ian (Ian F.), 1954-

Discovering requirements : how to specify products and services / Ian Alexander & Ljerka Beus-Dukic.
p. cm.

Includes bibliographical references and index.

ISBN 978-0-470-71240-5 (pbk. : alk. paper) 1. Requirements engineering. 2. Specification writing.

I. Beus-Dukic, Ljerka. II. Title.

TA180.A435 2009

658.5'038--dc22

2008049828

British Library Cataloguing in Publication Data

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

ISBN 978-0-470-71240-5

Typeset in 10.5/13 Palatino by Laserwords Private Limited, Chennai, India

Printed and bound in Great Britain by Bell & Bain, Glasgow

This book is printed on acid-free paper responsibly manufactured from sustainable forestry in which at least two trees are planted for each one used for paper production.

I dedicate this book to my father, Richard Alexander.
He was at once methodical and creative, positive and inspirational.

IFA

To my family and friends who encouraged and supported me.

LB-D

Contents

Acknowledgements **Foreword**

xv
xvii

Part I: DISCOVERING REQUIREMENT ELEMENTS	1
1 Introduction	3
1.1 Summary	4
1.2 Why You Should Read This Book	4
1.3 Simple but Not Easy	6
1.4 Discovered, Not Found	7
1.4.1 Many Different Situations	9
1.5 A Softer Process, at First	12
1.6 More than a List of ‘The System Shall’	16
1.6.1 A Network of Requirement Elements	16
1.6.2 Discovery as Search	18
1.7 A Minimum of Process: The Discovery Cycle	18
1.8 The Structure of this Book	20
1.8.1 Part I: Discovering Requirement Elements	21
1.8.2 Part II: Contexts for Discovery	22
1.9 Further Reading	22
1.9.1 Books on ‘Softer’ Approaches	22
1.9.2 Books on the Philosophical Background	23
1.9.3 Books on ‘Harder’ Approaches	24
2 Stakeholders	27
2.1 Summary	28
2.2 Discovering Stakeholders	28
2.2.1 Operational Stakeholders within ‘The System’	30
2.2.2 Stakeholders in the Containing System and Wider Environment	30

2.3	Identifying Stakeholders	37
2.3.1	From your Sponsor or Client	37
2.3.2	With a Template such as the Onion Model	37
2.3.3	By Comparison with Similar Projects	40
2.3.4	By Analysing Context	40
2.4	Managing Your Stakeholders	41
2.4.1	Engaging with Stakeholders	41
2.4.2	Keeping Track of Stakeholders	42
2.4.3	Analysing Influences	42
2.4.4	Prioritising Stakeholders	43
2.4.5	Involving Stakeholders	45
2.4.6	The Integrated Project Team	45
2.5	Validating Your List of Stakeholders	45
2.5.1	Things To Check the Stakeholder Analysis Against	46
2.6	The Bare Minimum of Stakeholder Analysis	46
2.7	Next Steps: Requirements from Stakeholders	46
2.8	Exercise	49
2.9	Further Reading	49
3	Goals	51
3.1	Summary	52
3.2	Discovering Goals	52
3.2.1	Worked Example: Goals for a Spacecraft	54
3.2.2	Worked Example: Goals for a Restaurant	57
3.2.3	Worked Example: Tram Goals and Trade-offs	59
3.2.4	Finding Solutions to Goal Conflicts	62
3.2.5	Contexts for Discovering Goals	63
3.2.6	The Negative Side	65
3.3	Documenting Goals	68
3.3.1	Drawing Goal Diagrams	69
3.3.2	Other Ways of Documenting Goals	69
3.4	Validating Goals	71
3.4.1	Things To Check Goals Against	73
3.5	The Bare Minimum of Goals	73
3.6	Next Steps	73
3.7	Exercises	73
3.8	Further Reading	74
3.8.1	Goals	74
3.8.2	The Negative Side	74
3.8.3	The i* Goal Modelling Notation	74
4	Context, Interfaces, Scope	75
4.1	Summary	76
4.2	Introduction	76
4.3	A ‘Soft Systems’ Approach for Ill-Defined Boundaries	77
4.3.1	You are Part of the Soft System you are Observing	78
4.3.2	From Stakeholders to Boundaries	79
4.3.3	Identifying Interfaces	83
4.3.4	Documenting Interfaces	84

4.3.5	Validating your Choice of Boundary	86
4.4	Switching to a ‘Hard Systems’ Approach for Known Events	87
4.4.1	The Traditional Context Diagram	87
4.4.2	Scope as a List of Events	87
4.4.3	Expressing Event-handling Functions	89
4.4.4	Strengths and Weaknesses of Context Diagrams	92
4.4.5	Validating Interfaces and Events	93
4.4.6	Things To Check Context and Interfaces Against	95
4.5	The Bare Minimum of Context	95
4.6	Next Steps	95
4.7	Exercise	95
4.8	Further Reading	96
4.8.1	Soft Approaches	96
4.8.2	Event-Driven Approaches	96
4.8.3	Writing Requirements	96
5	Scenarios	97
5.1	Summary	98
5.2	Discovering Scenarios	98
5.2.1	Interviews, storytelling	99
5.2.2	Scenario Workshops	101
5.2.3	Discovering Negative Scenarios	107
5.3	Documenting Scenarios	114
5.3.1	Index Cards, User Stories	115
5.3.2	Storyboards	116
5.3.3	Operational Scenarios	118
5.3.4	Use Cases	119
5.4	Summary	124
5.5	Validating Scenarios	124
5.5.1	Scenario Walkthroughs	124
5.5.2	Animation, Simulation, Prototyping	126
5.5.3	Things To Check Scenarios Against	127
5.6	The Bare Minimum of Scenarios	127
5.7	Next Steps	127
5.8	Exercises	128
5.9	Further Reading	128
5.9.1	Storytelling	128
5.9.2	Alternative Scenario Approaches	128
5.9.3	Running Scenario Workshops	129
5.9.4	The Principle of Commensurate Care	129
6	Qualities and Constraints	131
6.1	Summary	132
6.2	What are Qualities and Constraints?	132
6.2.1	A Rich Mixture	132
6.2.2	Qualities that Govern Choices	132
6.2.3	Constraints that Matter to People	133
6.3	Discovering Qualities and Constraints	133
6.3.1	Using Goals to Discover Qualities and Constraints	134

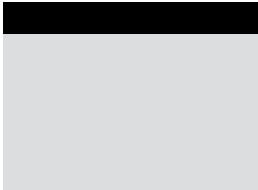
6.3.2 Stakeholder Analysis to Discover Qualities and Constraints	136
6.3.3 Using a Checklist to Discover Qualities and Constraints	136
6.4 Documenting Qualities and Constraints	141
6.4.1 Constraints	142
6.4.2 Development (Process) Qualities	146
6.4.3 Usage (Product) Qualities	147
6.5 Validating Qualities and Constraints	157
6.5.1 Things To Check Qualities and Constraints Against	158
6.6 The Bare Minimum of Qualities and Constraints	159
6.7 Next Steps	159
6.8 Exercises	159
6.9 Further Reading	160
7 Rationale and Assumptions	161
7.1 Summary	162
7.2 The Value of Rationale	162
7.3 Discovering Rationale and Assumptions	163
7.3.1 Asking Why	164
7.3.2 Looking for the word 'will' in vision statements, plans, etc	165
7.3.3 Rationalising a Set of Requirements	166
7.3.4 Inverting Risks	168
7.4 Documenting Rationale	169
7.4.1 Justification Text Field	171
7.4.2 Lists of Assumptions, Risks, Issues and Decisions	172
7.4.3 Traceability to Goals, Assumptions, etc	173
7.4.4 Rationale Models	178
7.4.5 The Goal Structuring Notation (GSN)	182
7.5 Validating Rationale and Assumptions	183
7.5.1 Rationale Walkthrough	184
7.5.2 Analysis of Traceability	184
7.5.3 Things To Check Rationale and Assumptions Against	186
7.6 The Bare Minimum of Rationale and Assumptions	187
7.7 Next Steps	187
7.8 Exercise	187
7.9 Further Reading	187
7.9.1 Discovering Assumptions	187
7.9.2 Reasoning	188
7.9.3 Modelling Rationale	188
7.9.4 Tracing to Goals	188
7.9.5 Goal Structuring Notation (GSN)	188
7.9.6 Satisfaction Arguments	188
8 Definitions	189
8.1 Summary	190
8.2 Discovering Definitions	190
8.2.1 Synonyms	191
8.2.2 Homonyms	193
8.3 Constructing the Project Dictionary	194
8.3.1 Acronyms	195

8.3.2	Definitions and Designations	195
8.3.3	Roles (Operational Stakeholders)	199
8.3.4	Data Definitions	201
8.3.5	Constraints as Data	202
8.4	Validating the Project Dictionary	204
8.4.1	Validating Data Models	205
8.4.2	Things To Check Definitions Against	206
8.5	The Bare Minimum of Definitions	206
8.6	Next Steps	206
8.7	Exercise	206
8.8	Further Reading	206
8.8.1	Definitions and Designations	206
8.8.2	Data Modelling	207
9	Measurements	209
9.1	Summary	210
9.2	Discovering and Documenting Acceptance Criteria	211
9.2.1	Acceptance Criteria for Behavioural Requirements	212
9.2.2	Acceptance Criteria for Qualities	216
9.2.3	Acceptance Criteria for Constraints	218
9.2.4	Verification Method	219
9.3	Validating Acceptance Criteria	222
9.3.1	Testing from Day One	222
9.4	Measuring Quality of Service (QoS)	223
9.4.1	Example Service: Office Carpeting	224
9.4.2	Two Opposite Approaches	225
9.4.3	A Spectrum of Service Approaches	226
9.4.4	Worked Example: QoS Measures for Food Preparation Services	228
9.5	Validating QoS Measures	230
9.5.1	Qualities of a Good QoS Measure	230
9.5.2	Will your QoS Measures Work?	231
9.5.3	Common QoS Measures	232
9.5.4	Validating QoS with Negative Scenarios	232
9.5.5	Things To Check Measurements Against	233
9.6	The Bare Minimum of Measurement	233
9.7	Next Steps	233
9.8	Exercise	233
9.9	Further Reading	233
10	Priorities	235
10.1	Summary	236
10.2	Two Kinds of Priority	236
10.3	Input Priority	237
10.3.1	Discovering Input Priority	237
10.3.2	Documenting Input Priority	241
10.3.3	Validating Input Priority	242
10.4	Output Priority	243
10.4.1	Discovering Output Priority	243

10.4.2 Documenting Output Priority	251
10.4.3 Validating Output Priority	253
10.5 Things To Check Priority Against	254
10.6 The Bare Minimum of Priorities	255
10.7 Next Steps	255
10.8 Exercise	255
10.9 Further Reading	255
10.9.1 Triage	255
10.9.2 Input Priority	256
10.9.3 Boston Matrix	256
10.9.4 Review Process	256
10.9.5 Life Cycles	256
Part II: DISCOVERY CONTEXTS	257
11 Requirements from Individuals	259
11.1 Summary	260
11.2 Introduction	260
11.3 Interviews	261
11.3.1 Planning an Interview Campaign	261
11.3.2 Planning Each Interview	267
11.3.3 Documenting Interviews	268
11.3.4 Validating Interview Findings	273
11.4 Observation and 'Apprenticeship'	274
11.4.1 Making Observations	274
11.4.2 Being 'Talked Through' Operations	276
11.4.3 Documenting Observations	277
11.4.4 Validating Observations	280
11.5 The Bare Minimum from Individuals	280
11.6 Exercises	280
11.7 Further Reading	281
11.7.1 Interviewing	281
11.7.2 Using Video	281
11.7.3 Observation	282
11.7.4 Tacit Knowledge	282
11.7.5 Standard Types of Systems Analysis	282
11.7.6 Informal Modelling Techniques	282
11.7.7 Philosophy	282
12 Requirements from Groups	283
12.1 Summary	284
12.2 The Goal of Group Work	284
12.2.1 Unique Capabilities	284
12.2.2 Obstacles	285
12.2.3 Mediating Group Work (on one site or many)	285
12.3 Workshops	286
12.3.1 Define Workshop Mission	286
12.3.2 Workshop Planning	287

12.3.3 Workshop Rehearsal	289
12.3.4 Workshop Setup	290
12.3.5 Workshop Recording	299
12.3.6 Validating Workshop Findings	302
12.4 Group Media	305
12.4.1 Project Wall	305
12.4.2 Project Website	306
12.4.3 Project Wiki	307
12.4.4 Modelling Tool	308
12.4.5 Requirements Management Tool	309
12.4.6 Groupware and Working at a Distance	310
12.4.7 The Role of Group Media	312
12.5 The Bare Minimum from Groups	314
12.6 Next Steps	314
12.7 Exercise	314
12.8 Further Reading	315
12.8.1 Workshops	315
12.8.2 Working in Groups	315
13 Requirements from Things	317
13.1 Summary	318
13.2 Requirements Prototyping	318
13.2.1 Purpose	319
13.2.2 Techniques	319
13.3 Reverse Engineering	330
13.3.1 From an Existing Product	330
13.4 Requirements Reuse	337
13.4.1 Type 1: Naïve Reuse	337
13.4.2 Type 2: Standardisation	338
13.4.3 Type 3: Product Lines	338
13.4.4 Tool Support for Reuse	338
13.5 Validating Requirements from Things	340
13.6 The Bare Minimum from Things	340
13.7 Exercises	340
13.8 Further Reading	340
13.8.1 Prototyping	340
14 Trade-offs	343
14.1 Summary	344
14.2 Optioneering: The Engineering of Trade-offs	344
14.2.1 The Requirements-First Life-Cycle Myth	344
14.2.2 An Optioneering Life Cycle	345
14.2.3 The Optioneering Process	350
14.2.4 Selecting the Winning Option	352
14.2.5 Optioneering with PCA: A Worked Example	360
14.3 Validating your Trade-offs	367
14.4 The Bare Minimum of Trade-offs	367
14.5 Next Steps	367

14.6 Exercises	368
14.7 Further Reading	369
14.7.1 Trade-offs	369
14.7.2 Statistics	370
14.7.3 PCA	370
14.7.4 Weighting Approaches	370
14.7.5 Analytic Hierarchy Process (AHP)	370
14.7.6 Quality Function Deployment (QFD)	370
14.7.7 Questions, Options, Criteria (QOC)	371
15 Putting it all Together	373
15.1 Summary	374
15.2 After Discovery	374
15.2.1 Everything Depends on the Requirements	374
15.2.2 Principles for the Requirements Chef	375
15.3 The Right Process for your Project	376
15.3.1 Case Study: A Retail IT Project	377
15.3.2 Case Study: Transport Planning	379
15.3.3 Requirements-Driven Project Management	381
15.4 Organising the Requirements Specification	385
15.4.1 Template	385
15.4.2 Levels	385
15.4.3 Can Use Cases Do Everything?	386
15.4.4 Organising Product Functions	386
15.4.5 Traditional 'Shalls'	387
15.4.6 Relating Requirements of Different Types	388
15.4.7 Conflicting Needs for Requirement Organisation	390
15.4.8 The Benefit of Requirements (Traceability) Tools	390
15.4.9 An Alternative View: Competing Approaches	391
15.5 The Bare Minimum of Putting it all Together	394
15.6 Further Reading	394
15.6.1 Choosing and Tailoring Development Life Cycles	394
15.6.2 Managing Projects From Requirements	395
15.6.3 Classics for Inspiration and Reflection	395
15.6.4 A Look Ahead	396
APPENDIX A: Exercise Answers and Hints	397
APPENDIX B: Getting the Level Right	405
APPENDIX C: Tools for Requirements Discovery	411
APPENDIX D: Template	423
Bibliography	429
Glossary	433
Index	445



Acknowledgements

We are enormously grateful to our team of expert reviewers from around the world: Joy Beatty; Keith Collyer; Adie Ditchburn; Marko Dukic; Robin Goldsmith; Frank Houdek; Ellen Kustrin; Soren Lauesen; Martin Mahaux; Sasikumar Punnekkat; Narayanan Ramanathan; Pete Sawyer; Gerhard Schuster; Rune Stamselberg; and Ralph Young. We thank them for their careful reading and their perceptive comments. They have immeasurably improved the book.

This book owes an unquantifiable debt to the whole Requirements community, the RE'xx conferences, and the committee and members of the BCS Requirements Engineering Specialist Group.

We are very grateful to the guest authors who generously contributed to the book: Joy Beatty; Robin Goldsmith; Ellen Gottesdiener; Frank Houdek; Soren Lauesen; Martin Mahaux; and Ralph Young. Requirements discovery is always a team effort, and the book is much enriched by the diversity of their skills and experience.

We would like to thank the students at the University of Westminster.

Special thanks to Ian's family for their support and understanding through the time that he was working on this book.



Foreword

As information technology grows in power and reach, systems become more ambitious and more tightly integrated. The discipline of defining - or, perhaps, engineering - requirements becomes broader and deeper: it now embraces almost every project that aims to define a human or technical need and to find a means to satisfy it.

Inevitably, the requirements engineering community is a very broad church. Its adherents include practitioners and thinkers of all inclinations, ranging from those who relish challenges in the softest human and social terms to those who prefer to grapple with the hard-edged technical intricacies of defining the details of required software behaviour in any problem context, from a physical device to a business organisation.

Ian Alexander and Ljerka Beus-Dukic are teachers and practitioners, very comfortable in the centre and softer areas of the discipline. They are interested, above all, in the process of discovering requirements in realistic human contexts, and their book reflects and conveys the scope of their interest, experience and wisdom. It is filled with practical advice, based on both experience and thought. They offer insights and help in the vital pragmatics of eliciting and clarifying the needs of multifarious stakeholders, and of balancing their conflicting interests and goals. They are strong on the social skills that the practitioner needs in order to identify what is really required, and on the techniques of trading off stakeholders' requirements against one another and against what is achievable in practice.

The book is annotated with aphorisms drawn from a multitude of sources – some from the ever-growing literature of requirements engineering, some from thinkers as diverse as Winston Churchill and Walter Vincenti, a wonderfully insightful aeronautical engineer. Every human milieu provides illuminating examples: the provision of a meal, the construction of an Indonesian outrigger boat, the use of a hand-held wireless device on a mountain hike – all furnish both insight and amusement.

The book has an interesting matrix organisation. Earlier chapters focus on the elements of requirements – stakeholders, goals, scenarios and priorities. Later chapters focus on the contexts and sources – individuals, groups and artifacts – from which these elements can be elicited and the full requirements fashioned. Two final chapters offer advice on trade-offs and how to put it all together in the eventual implementation project.

This book is not only of practical value. It's also a lot of fun to read. I particularly liked one remark that chimed happily with my own prejudices: 'We believe that for any task, whether you are learning from a book or doing practical work on a project, you need to balance periods of action with periods of reflection.' I agree wholeheartedly. And I'm sure you will agree, as you enjoy this book, that its authors have followed their own advice to admirable effect.

Michael Jackson

PART I

Discovering Requirement Elements

Requirement Elements	Priorities
Discovery Contexts	Measurements
	Definitions
Introduction	Rationale
From Individuals	Qualities and Constraints
From Groups	Scenarios
From Things	Context, Interfaces, Scope
Trade-Offs	Goals
Putting it all Together	Stakeholders

This part of the book describes what you need to discover – the different requirement elements – together with practical techniques to create, document and validate them. On most projects, you will probably need to use most of these at some stage. The requirement elements are described roughly in the order you are most likely to use them.

CHAPTER ONE

Introduction

*Fare buon vino è semplice ma non facile.
(Making good wine is simple but not easy.)*

Italian farmer

Requirement Elements	Priorities	Measurements	Definitions	Rationale and Assumptions	Qualities and Constraints	Scenarios	Context, Interfaces, Scope	Goals	Stakeholders
Discovery Contexts									
Introduction									
From Individuals									
From Groups									
From Things									
Trade-Offs									
Putting it all Together									

1.1 Summary

Requirements work is simple but not easy: it is a craft, which requires skill. Fortunately, that can be learned.

Requirements are discovered by the use of appropriate inquiry techniques. They are not sitting about, waiting to be ‘captured’. Each of the many different situations demands a varied set of techniques.

When requirements are undefined, a project knows neither what it needs to do nor how big a problem it faces. A ‘soft systems’ process that can handle undefined problems is required, at least at first.

Requirements are more than a list of statements of what a system must do. A better approach is to define a network of related elements, including such things as definitions, goals, rationale and measurements.

This book is organised into two parts:

- Part I, Discovering Requirement Elements;
- Part II, Discovery Contexts.

The book’s chapters form the columns (Part I) and rows (Part II) of a matrix that appears at the beginning of each chapter. Any given discovery activity will involve an element from Part I being discovered in a context from Part II. Each chapter in Part I, therefore, cuts across the contexts; each chapter in Part II cuts across the elements.

Requirements work involves a discovery cycle that entails: discovering with stakeholders; documenting; validating with stakeholders. Each chapter in Part I, Discovering Requirement Elements, is structured according to the discovery cycle.

This book is designed to be entirely practical but, for success, you need to read and reflect as well as work. Books for further reading are suggested.

1.2 Why You Should Read This Book

This book is about what you have to do to discover requirements, whatever kind of business you are in. There are plenty of books and tools designed to help you organise and manage your requirements once you have them, but actual coverage of the critical early stages—of discovery—is far patchier.

The book looks in turn at all the basic elements of requirements, and then at how to discover them. Strangely enough, it seems to be the first book to do this systematically; few popular texts say much about basic requirement

elements like goals, qualities, constraints and rationale and, while there's plentiful coverage on use cases, practical advice on stakeholder analysis and trade-offs is sparse.

Requirements communicate needs from stakeholders to developers on a development project. A development project is a time-limited undertaking with a single purpose: to create the new thing—the product or service—that is named in its mission.

So, if you are involved in a development project, whether as engineer or developer, product manager or business analyst, team leader or project manager, and you need to find out what your project should be doing, this book is for you. We hope the book will be helpful to marketing people, too; after all, marketing discovers requirements, even if it isn't usually expressed in those words.

Words like 'requirement' slip into and out of fashion. Terms like 'systems analyst' have fallen into disuse; new ones like 'business analyst' are becoming popular. In this book, we have tried to focus on the essence of each activity. For instance, we consider different ways to tell the story of what a product should do, including user stories, operational scenarios and use cases. All have their merits but the basic element is the human one of storytelling. We have tried to make this book describe the unchanging core of the work of discovering business needs, so we hope you will find it helpful whatever methodology (and terminology) you prefer.

Because this book's focus is purely practical, we also hope it will be useful to students and lecturers who want to learn about requirements work in industry. The book is arranged in chapters that we have tested as course materials for undergraduate software engineering students. We have included short exercises that could form the basis of tutorial or project work.

Every chapter describes how to apply its techniques to project situations, with practical tips and examples. We have tried hard not to assume prior knowledge on your part, though you will certainly get more from the book if you have project experience. If you are still a student, you will probably find some of the topics obscure until you start trying to discover requirements for yourself. We hope, however, that the examples and illustrations from our own experience will give you an idea of why each technique is useful in industry.

We are aware that busy practitioners have many demands on their time. We have carefully organised this book to make it quick to use.

We believe that for any task, whether you are learning from a book or doing practical work on a project, you need to balance periods of action with periods of reflection. The chapters therefore contain 'grey boxes' that encourage you to reflect on the main text: to read further; to understand the background of an idea or technique; and to improve your own practice.

1.3 Simple but Not Easy

Perhaps your first thought about requirements work is that it is simple, and that you just write down what you need.

Perhaps your experience of trying to write down what you need is that it is not easy, and that you did not get what you wanted.

'Simple but not easy'. Why is that? Things that look simple but are not easy include many traditional skills and crafts.

On a trip to Indonesia some years ago, Ian was lucky enough to be able to watch some traditional boatbuilders at work. They had no workshop other than the beach, and no tools other than hand-adzes, hand-drills and wooden mallets. They used nothing except wood, with moss to fill in the gaps between planks. How did they make the planks fit? How did they know what shape to make each plank? How did they select the wood? They tapped and scraped, a little at a time. They put wooden dowel pegs in the holes in the plank they had just fitted. Then they carefully lined up the holes in the newly shaped plank over the dowels, and gently tapped the plank into place until it was snug and watertight. Each operation was beautifully simple. The final result was a strong, seaworthy boat, like the one illustrated below.



A hand-built outrigger boat, Flores, Indonesia

What are the steps involved in making a set of requirements watertight? How do the parts fit together? How do you create and check each part? What skills do you need for each part? These are the questions this book tries to answer.

1.4 Discovered, Not Found

*Requirements cannot be observed or asked for from the users,
but have to be created together with all the stakeholders.*

Vesa Torvinen

People have talked for years about requirements capture, gathering, trawling, or elicitation. The words paint attractive pictures of requirements as things you could imprison in a net like fishes or butterflies, harvest like berries, or coax from taciturn experts (Figure 1.1).

But requirements are discovered in a wide range of ways. They are more often created by collaborative work than casually found. As Hugh Beyer and Karen Holtzblatt (1998) write in their book *Contextual Design* [1]: 'Requirements and features don't litter the landscape out at the customer site.'

Often, people do not know what they want either, so it isn't enough just to ask them. You might have to show them prototypes (Chapter 13), devise tests, or work steadily with people towards defining what they want, using the techniques described in Part 1 such as goal and context modelling.

Discovery may also sound somewhat too easy. Like most words, it has different meanings. By discovery, we do not mean stumbling onto things by chance. When teams of scientists discover the quark or the double helix or giant magnetoresistivity or the golden-mantled tree-kangaroo, it is generally not by accident. Discovery, however surprising and delightful the actual moment of realisation, comes as the result of a deliberate search. That search is guided by theory, which itself grows and adapts to whatever is discovered, and is built on evidence.

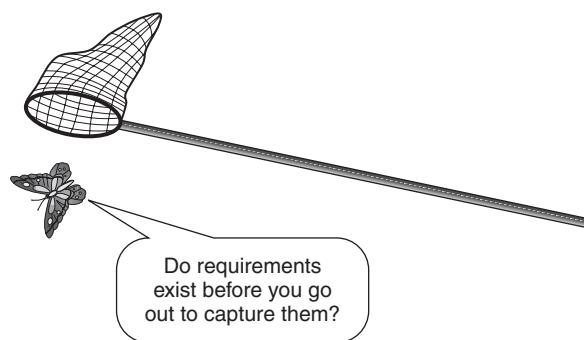


Figure 1.1: Capturing requirements.

A Moment of Discovery

A development project is not a controlled scientific experiment. It is very difficult to prove—and rare even to become aware of—what a project would have been like had requirements not been discovered by the normal systematic use of techniques like those described in this book.

One such moment of conscious discovery came while I was working on a railway project. I was in a workshop, walking through the ‘big picture’ scenario of how a new train control box was to be brought into service. A conversation ensued, which went something like this:

Me: ‘And then you’ll be testing the box on the train during a quiet time of day, in between in-service trains?’

Client: ‘Um, well, no, we can’t run it on the track while other trains are in service in case it fails and causes delays . . . it’s against the regulations until it has its safety certificate.’

Me: ‘Ah, I see. So you’ll be running tests during the night, then, in engineering hours?’

Client: ‘Oh no, we won’t be able to get enough hours for that. Engineering hours are in demand for all sorts of other purposes. Anyway, most nights the power is turned off to enable the line to be cleaned and the track to be maintained.’

Me: ‘Could you maybe just run it around the depot, then?’

Client: ‘We could . . . but there are no signals there, so we can’t test most of the functions of the box . . . in any case, we’re only allowed to run trains in slow manual (5 mph) in the depot.’

Me: ‘Well, presumably you have a nice loop of test track with a couple of signals on it so you can try things out properly?’

(Here the workshop participants shifted uncomfortably in their seats and looked at each other nervously.)

Client: ‘That would be nice, but we don’t have one.’

(A penny dropped in a slot in my brain.)

Me: ‘In that case, we’ll have to build a simulator. We can tell the contractor to build some software to tell the train control box it is leaving the station, coming to a signal, and arriving at the next station.’

(I was still not prepared for what came next.)

Client: ‘Yes, but every station is different. Some have track sloping down, so the train tends to accelerate as it comes in, so the signals have to be further apart on the approach; others slope up, so the signals can be closer together. Some are underground and so are dry; others are above ground and can get wet or icy, so again we allow more distance between the signals there.’

Me: 'In that case, there is no choice: we need a whole-line simulator. Are there plans of the track and signals on the line?'

This discovery was rare, in the sense that it was sudden and large, and everyone realised that a missed requirement had been discovered. Presumably, the omission would finally have been noticed when the test campaign was planned. That would have been most inconveniently late in the project. It would certainly have cost millions of pounds and caused many months of delay.

The need for the simulator had been missed by a specification process that largely followed tradition; for example, there were written requirements and state transition diagrams. But the project's wider context and its bigger picture scenarios had not been thought through.

If there is a lesson to be learned from this, it is that projects need to pay attention to discovering their requirements, using a battery of complementary techniques that include, for instance, not just analysis but also scenarios. That way, the chance of anything large falling through the net is greatly reduced.

Discovery, then, means many different things, such as:

- looking at the evidence;
- being open to new ideas;
- applying creative effort;
- working as a team;
- asking questions that focus the search;
- intending to find particular kinds of thing;
- fitting whatever is found into a reasoned framework;
- relating whatever is found to similar discoveries.

In these senses, requirements discovery is what you need to create a solid foundation for your project.

1.4.1 Many Different Situations

There is no single technique for discovering universal requirements. Projects are of very different kinds, so requirements have to be discovered in diverse ways, most often by working with the appropriate people (Table 1.1).

Projects range from the strictly formal and contractual (as in a custom development or subcontract), to the brief and sketchy (as in small-scale software redevelopment in what is wrongly called software maintenance).

Table 1.1: Where requirements come from.

Type of project	Source of requirements
In-house development by an organisation's IT department	People within the organisation: software users, managers and the IT department itself. No contract ('We can't sue our IT department').
Product or service development for the mass market	Marketing, product management (on behalf of the public and the company); technical experts (specialists) in different disciplines.
Custom development for a single client	Business requirements from the client's technical people; terms and conditions from its commercial side.
Commercial off-the-shelf (COTS) package purchase/tailoring	Package selection, often done in-house by matching needs to package capabilities; tailoring, as for either in-house or custom development .
Subcontract e.g. within a product development	Prime contractor, by derivation from the system requirements and design. Most—possibly all - of this work is analysis rather than discovery as such, though gaps may reveal missed business requirements.
Maintenance support for earlier custom development	System/software users within the client organisation (via change requests received once the product is in service—these are often very informal); problem reports.

Perhaps the projects in greatest danger from poor requirements work are those that seem fairly small and simple, but turn out to contain hidden complexities.

Guest Box: Robin Goldsmith on REAL Business Requirements

Robin F. Goldsmith, JD, author of the book *Discovering REAL Business Requirements for Software Project Success* (2004) [6], describes a key concept from his experience of requirements discovery.

Requirements are NOT all the Same

It's not only that requirements must be discovered rather than gathered, but REAL requirements also are not specified. One of the common major

sources of difficulty starts with the failure to distinguish between two fundamentally different types of requirements.

- **Business requirements** are the REAL requirements which provide value when they are met, satisfied, or delivered. Business requirements are from the perspective of, and in the language of, the business or user. I am using the term 'business' broadly - for work or personal, for profit or not. REAL business requirements are conceptual and exist within the business environment, which is why they must be discovered. They are business deliverable *whats* that provide value by serving business objectives through solving problems, taking opportunities, and meeting challenges. There usually are *many possible ways to accomplish them*.
- **Product requirements** are from the perspective of, and in the language of, a human-defined product which is *one of those possible ways presumably to accomplish* the business requirements. Since these often are phrased in terms of the product's external functions, they are also called 'functional specifications', which embraces the illusory distinction from 'non-functional requirements.' Humans specify designs. Product requirements are design, which is not limited to engineering technical detail. They provide value if and only if they meet the REAL business requirements.

People, including most fellow requirements authors, usually are referring to product requirements when they use the term 'requirements.' Many do use the term 'business requirements' to mean vague high-level requirements, often just purposes and objectives, which they mistakenly believe decompose into detailed product requirements. Widely-accepted conventional wisdom also holds that creep—changes to requirements after they've supposedly been defined—is due to unclear and untestable product requirements.

In fact, much of creep occurs because product requirements, regardless of how clear and testable they are, do not meet the REAL business requirements. The primary reason is that the REAL business requirements have not been defined adequately and in detail, primarily because people concentrate mainly on product requirements.

The Problem Pyramid™ is a powerful tool that helps draw these distinctions and guide discovery of high-level REAL business requirements. These then must be driven down to sufficient detail to make them adequately clear and testable. At every hierarchical level of detail, REAL business requirements are business deliverable *whats* that provide value when delivered. They never are *hows*. The difference is not level of detail.

The *Problem Pyramid*™ consists of six steps which must be performed in sequence:

1. Identify the **REAL Problem, Opportunity, or Challenge** which provides REAL value when addressed appropriately. This is exceedingly difficult. Failure to identify the REAL problem leads many requirements definitions astray from the start.
2. Identify the **Current Measure(s)** which tell us that the Problem is a Problem. Defining measures is part of defining the REAL Problem. Failure and inability to define measures often indicates that the Problem has been not defined correctly.
3. Identify the **Goal Measure(s)** which tell us the Problem has been addressed. Meeting the Goal Measures(s) provides value.
4. Identify the **Cause(s)** of the Problem. Causes are the *As Is* process producing the Current Measures. One doesn't solve a problem directly but rather one solves it by identifying and addressing its causes.
5. Define the business deliverable *whats* that when delivered will provide value by achieving the Goal Measure(s). These are the *Should Be* process and are the **REAL Business Requirements**.
6. Specify a product **Design** *how* to satisfy the REAL Business Requirements.

Reproduced with permission of Robin F. Goldsmith.

1.5 A Softer Process, at First

Fortunately, people (in universities and in industry) have developed a broad understanding of how to go from a situation where nobody even knows what problems ought to be solved, to a position where everybody agrees on the problem that is to be solved. After that point, a definite product or service can be designed to solve the problem effectively, and a more procedural process of 'requirements management', supported by database tools, can take over from discovery or 'requirements definition'.

To reach that point, a 'softer' process than product design, focused more on people than on products, is needed (see Chapter 4). That process iterates the nature of the business, its issues and ambitions, until a clear decision—say, to develop a product—emerges.

Once preliminary development funding, at least, is made available, work can begin on discovering the requirements and defining the scope. Later, when

the shape of the product or service is sufficiently well-defined, harder and more definite modelling approaches can take over. Academics sometimes call these the ‘early’ and ‘late’ requirements phases. A book on discovering requirements must naturally focus mainly on the earlier, creative aspects of requirements work. As Kotonya and Sommerville (1998) put it in their textbook *Requirements Engineering* [5]:

‘Structured methods of requirements analysis ... are not particularly useful for the early stages of analysis where the application domain, the problem, and the organisational requirements must be understood. They are based on “hard” models ... [which] are inflexible and focus on the automated systems.’

The softer processes invariably consist of collaborative techniques that involve a group of people with different backgrounds and experiences. Developers tend to call people in all other roles ‘users’. Consultants know the importance of ‘clients’ or ‘customers’. Academics and politicians talk about ‘stakeholders’, a safely neutral word. (Stakeholder analysis is described in Chapter 2.) Consultants and developers, far from being experts, are often the outsiders in whichever domain or area of expertise (such as finance, telecommunications or aerospace) is being addressed.

People working on requirements have to learn *both* about the domain and about what the stakeholders feel is the problem; requirements touch on both ‘soft’ and ‘hard’ systems work (Figure 1.2; and see Chapter 4). For consultants, analysts, developers and (in an effective requirements process) other stakeholders alike, this really is discovery.

There are any number of job titles that cover all or part of this work. ‘Systems engineer’ applies mainly to more hardware-centred industries, whereas ‘business analyst’ seems to apply mainly to IT systems, so perhaps ‘requirements analyst’ is a better term for our purposes. The rather loosely defined relationships of some of these professions to the areas of work we are discussing are indicated, very approximately, on Figure 1.2.

There are numerous soft¹ approaches. Some span all of development (usually of software); others are meant mainly for understanding and reorganising whole businesses. For instance:

- **Contextual design** [1], devised by Hugh Beyer and Karen Holtzblatt (1998), starts by exploring the work context, models influences in the ‘culture’ of a workplace, and continues right through to user interface design.
- **Soft systems methodology** [2], pioneered by Peter Checkland (1990), is generally applied at a strategic level, spanning many domains and any

¹‘Soft’ here means that the work is ill-defined, involving many partially known concerns, not susceptible to deterministic approaches. It does not mean ‘about software’.

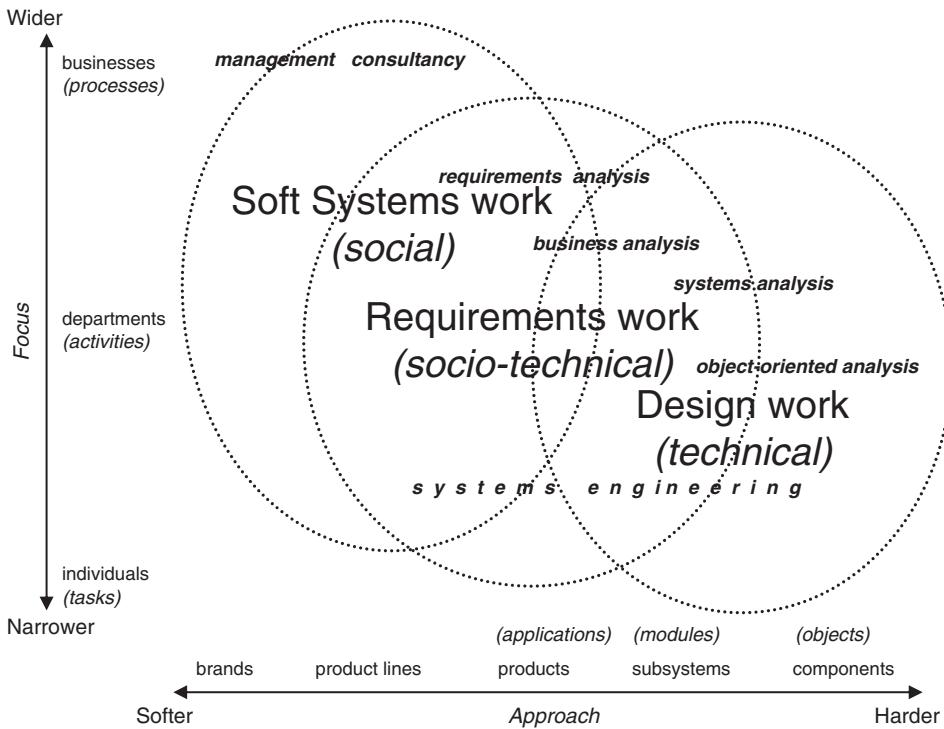


Figure 1.2: Requirements discovery has to span the divide between ‘soft’ and ‘hard’, between stakeholder problems and technical solutions.

number of possible future products, but in the main stopping short of individual product specification.

- Similar ideas in strategic planning, business analysis and other fields attempt to understand what a business is trying to achieve, and redesign how it works. For instance, James Dewar’s (2002) **Assumption-based planning** [3] offers a fresh insight into discovering assumptions.

The techniques advocated in such processes are not difficult and do not need elaborate tools, but together they offer the requirements analyst the chance of creating better requirements.

None of this predetermines the development life cycle that you should adopt. You can think of the requirements cycle as a smaller wheel inside whatever life cycle you use.

Many organisations write a set of ‘stakeholder requirements’ at the point in their life cycle where they know they want to create a system but have not chosen its design. They then evaluate alternative designs and select the one they

Requirements ‘Engineering’?

Many years ago, the inventor of the DOORS requirements tool, Richard Stevens, phoned me up and said he was working on a tool to do requirements engineering.

‘How can you engineer a requirement?’, I asked in astonishment. I’d only seen requirements as a list of very dull statements like, ‘The system shall enable the user to edit the “Name” field.’

What did that possibly have to do with engineering? Richard asked me to suspend my disbelief. He showed me what the tool did, and enthusiastically painted a picture of his vision.

The tool sold very well and did an excellent job of managing requirements on some incredibly big systems in some amazingly complex business environments:

- One firm used it via dedicated satellite links to share confidential requirements on sites on opposite sides of the Atlantic Ocean.
- Another used it to monitor numerous contractors, who were each implementing a different part of a huge specification, via an intricate process of locking slices of the database and passing the slices backwards and forwards as files.

It was fascinating to work with. It involved tools, modelling, database design, customisation with scripts (which sometimes meant weeks of programming), training and data handling. This was certainly software engineering. But were we actually engineering the requirements?

I think now that we were engineering the *management* of the requirements. Other people worked on the requirements as such, in an altogether ‘softer’ way: much less like engineering; much more like discovery.

will use. In the light of that choice, they write a set of ‘system requirements’,² usually much more detailed than the stakeholder requirements, describing what the (known design of) system will have to do. If the system is large, this is followed by a further stage of design, breaking the system down into several subsystems, each specified in its own set of ‘subsystem requirements’. The design of life cycles is outside this book’s scope, but Appendix B discusses how to identify the level to which a requirement properly belongs. Since it is easier

²These are often requirements relating only to the product under design so the name ‘system’ is not ideal, but it is widely used. Where the requirements cover issues such as staff training as well as hardware and software, system is certainly the right word. See Chapter 4 for discussion.

to think about details than about the big picture, people often state ‘system’ or ‘subsystem’ requirements (prematurely) in their stakeholder requirements.

1.6 More than a List of ‘The System Shall’s’

The old definition of a requirement as a separately verifiable contractual statement is still valid, but is not very useful during requirements discovery.

It is not possible to start writing formal requirements until you know who the stakeholders are, what their goals are, what the context is and so on. These things are not ‘requirements’ in the old, narrow sense, but defining them does take you up to the point where you know what problem you want to solve, and can communicate that to a supplier. In this broader sense, goals and scenarios and so on certainly form the requirements; individually, we can call them ‘requirement elements’.

‘The requirements’ in the broad sense means a network of interrelated requirement elements: a requirement that satisfies a goal, is justified in a rationale model, using terms defined in the project dictionary, etc. This is a richer structure than an old-fashioned list of statements, and it fulfils its purposes better. Part I of this book describes these requirement elements, and ways of discovering each of them.

1.6.1 A Network of Requirement Elements

Just as no single technique is sufficient for discovering requirements, so there is no single way of documenting a requirement that is suitable for every situation. Instead, there is a set of commonly occurring elements that together can define what is wanted. Figure 1.3 shows these, together with some relationships between them.

Many other interrelationships are possible. For example, a scenario may reveal the need for an interface; an assumption may justify a measurement; a stakeholder may be responsible for some constraints.

These elements should largely be familiar, as they are simply tidied up versions of concepts that everyone uses in daily conversation: roles, stories, events, goals, reasons and so on. Indeed, it seems that many of them are fundamental to the structure of every human language, if the ideas advanced in Steven Pinker’s (2007) *The Stuff of Thought* [7] are correct. That Pinker really is close to the truth can be seen in the way that different methods select subsets of these elements (see Chapter 15 for further evidence).

Figure 1.3 is a coarse-grained³ model of a possible organisation of the requirements information within a project. Each box in the diagram (except

³In a finer grained model, more elements would appear. For instance, ‘scenario’ would resolve into ‘scenario step’, ‘exception’, and so on.

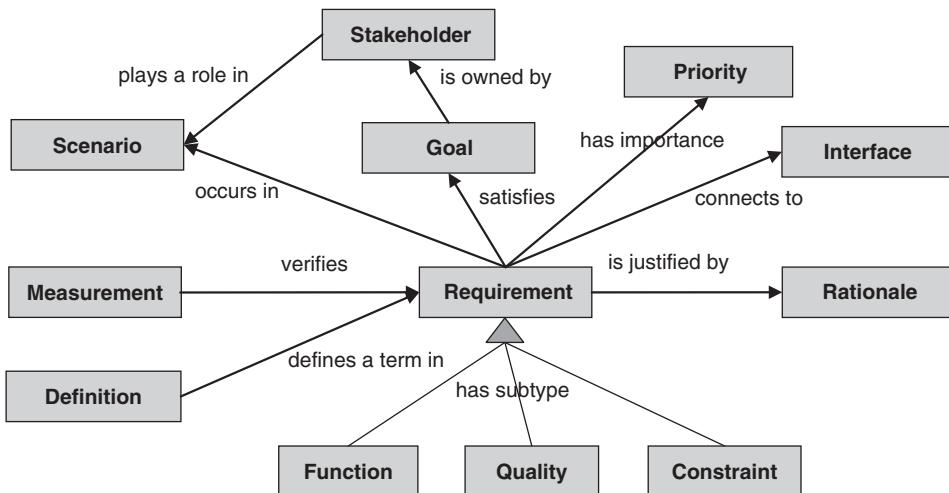


Figure 1.3: Requirements specification as a network of elements.

'requirement' itself) names a requirements element. All but one of the element boxes correspond directly to chapters in Part I of this book. The exception is 'function', which is covered by Chapters 4 and 5. The writing of functions is also described in Ian Alexander and Richard Stevens' (2002) *Writing Better Requirements* [12]. Interfaces are both design (of the containing system) and requirements (on the contained product). Since interfaces may be known in advance, they show that the idea that requirements come before design is not necessarily true.

Different projects vary considerably in terms of the importance of the different requirement elements. In practice, therefore, projects sometimes entirely omit some of the elements shown in Figure 1.3. For example, a project with a simple stakeholder structure might omit 'stakeholder analysis'. The resulting lack could be compensated for by additional explanation of stakeholder issues in 'scenarios', 'definitions' or 'rationale'. For more on tailoring a requirements approach for your own project, see Chapter 15.

The information model drawn in Figure 1.3 only covers requirements. Other related elements in a project's information model include plans, risks, issues/decisions and tests. These are important to projects, will have connections to the requirement elements, and may be appropriately recorded in a requirements database (see Appendix C) but are outside the scope of this book.

Figure 1.3, and indeed the chapter structure of this book, can be seen as a customisable template for organising the requirements on your project. For convenience, a generalised template is provided as a starting point in Appendix D.

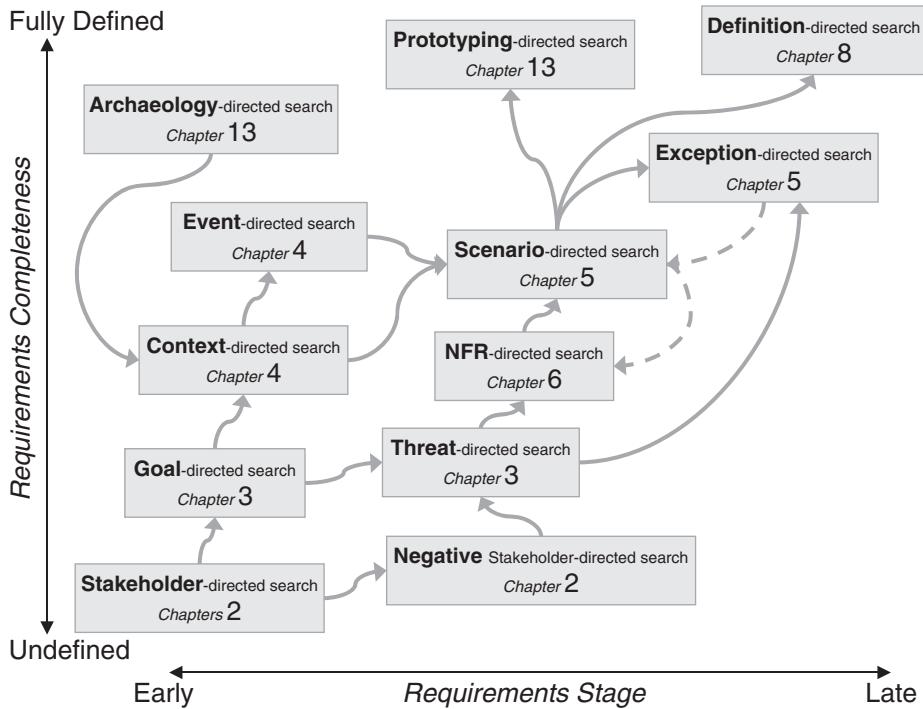


Figure 1.4: Requirements discovery as search, directed by the requirements elements so far discovered.

1.6.2 Discovery as Search

A good way of thinking about discovery is as a search (Figure 1.4). You can use different techniques, related to the requirement elements defined in Part I of this book, to direct your search at different stages. In other words, the structure of what you know drives what you discover next. The better you organise your knowledge of the requirements, the better you can discover what is really needed.

For example, you discover a goal. Then you can use a scenario to explore how to achieve that goal. Then you can search for exceptions that can occur at each step of that scenario.

1.7 A Minimum of Process: The Discovery Cycle

A generic discovery process can be drawn as a simple inquiry cycle (Figure 1.5).

An inquiry cycle is more or less what it says: a cycle of activities, carried out by a team, to enable them to inquire effectively into some subject (such as what precisely should be developed as a product).

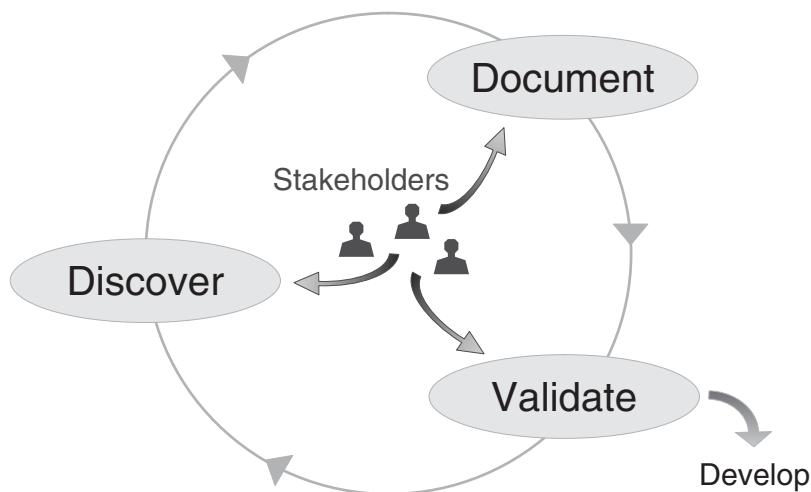


Figure 1.5: Inquiry cycle for creating requirements.

What all inquiry cycles have in common is a period of action followed by a period of reflection (Heron 1996) [4]. Both are necessary: action to get on with the work; reflection to consider whether the work is complete, or going in the right direction.

Agility

It is fashionable nowadays to use the word ‘agile’ a lot. On some kinds of project, you can discover a handful of requirements, implement them in a product at once and then go back to discover a few more. This approach can be very helpful in some situations, as it gives customers something to look at, and this may prompt them to tell you some new requirements that they hadn’t known about before.

Agility is not really new; people used to call it ‘rapid prototyping’, ‘iterative development’, ‘rapid application development’ and other such complicated names. Good project managers and project teams have always practised it.

Rapid iteration does not in any way absolve you from discovering the requirements. Agile or not, the decision depends on your situation. But, discovering requirements is crucial.

For example, if you hold a workshop to discover scenarios (Chapter 5), you are immersed in action. Afterwards, quietly analysing models of those scenarios to document the workshop’s findings, you have time to reflect on

what you have found. Then, armed with a fresh list of questions, you go back to the relevant stakeholders and validate your findings. This may lead you to further discovery activities.

However, the techniques for discovering, documenting, and validating a set of scenarios (for instance) are not necessarily the same as those for discovering goals, or risks, or stakeholders. Also, it's no good waiting until the end of the requirements phase before you try to validate each finely polished requirement. You need to do that right away, checking each little discovery as you go along. So there isn't one enormous requirements process, but many little inquiry cycles. You could call this 'agile requirements discovery'. For example, Mike Cohn's (2004) *User Stories Applied* [8] offers a fully worked out agile approach (for software). Accordingly, each chapter in Part I covers the discovery, documentation and validation of the element it discusses.

Apart from that very basic process, this book's focus is essentially on what you actually need to do on your project. It does not propose some new overarching process, framework or methodology, because:

- Every project is different; standardised processes can only describe the average or typical case (and a few common variations). The final chapter of this book describes how to tailor your own requirements process from the building blocks described in the earlier chapters.
- The actual techniques don't have to be used in some official, fixed process or method at all. You can scribble a goal model during an interview, or while reflecting afterwards on the train home. You can do a little stakeholder analysis and note down a few scenarios, and then get on with something else. If this solves a problem on a project, that's fine.
- What is missing isn't theory, but practice: projects creating their requirements simply and carefully. This book describes straightforward ways of doing that.

1.8 The Structure of this Book ---

This book is divided into two main parts:

- Part I describes what you need to discover – different requirement elements—with practical techniques to create, document and validate them. On most projects, you will probably need to use most of these at some stage. The requirement elements are described in roughly the order you are most likely to use them. They do not form a single mandatory process or anything like that. You have to use your common sense and develop the experience to apply those that will work best on your project.
- Part II describes the contexts where you can effectively discover requirements. These include the traditional environments in which consultants

meet stakeholders – interviews and workshops – as well as consulting the public, observation, ‘reverse engineering’ and reuse. It also describes how to make the transition from discovery to using the requirements to drive your project. This includes trading off what is wanted against what can be achieved.

Each chapter in Parts I and II is structured to help the reader, with:

- a list of questions and answers;
- a goal;
- a summary;
- the main text, illustrated with diagrams, tables and photographs;
- a list of bare minimum activities;
- next steps;
- exercises, with hints on answers at the back of the book;
- further reading.

1.8.1 Part I: Discovering Requirement Elements

Part I describes the things you need to find out, a step at a time. They may be created directly by working with stakeholders in the contexts described in Part II, or they may be analysed quietly in between meetings with stakeholders.

Each chapter describes one element (e.g. scenarios), with worked examples, containing sections saying how to:

- discover it;
- document it;
- validate it for completeness, correctness, and consistency.

The elements of Part I include:

- a list of the stakeholders, and their influences on each other;
- goals, including the negative side;
- context, interfaces and scope of the work, and of possible future products, including a list of the events to be handled;
- stories, scenarios and use cases that describe how the product could be used to deliver the wanted results;
- qualities and constraints that any acceptable product must meet;
- rationale and assumptions, arguing the case for (and against) decisions, such as choosing particular requirements and design options;
- technical terms, data definitions and roles used in a project;
- acceptance criteria and verification methods, or qualities of service, defining how you will know that the requirements have been met;

- priorities, both input (desired by stakeholders) and output (agreed by the project for a given phase of development).

The descriptions in each case are enlivened with brief accounts of practical experience.

1.8.2 Part II: Discovery Contexts

Part II describes practical ways for developers to work together with stakeholders to discover the requirements. The contexts it describes include:

- interviews;
- workshops;
- observation of or participation in the work;
- ‘reverse engineering’—discovery from existing systems;
- reuse, where requirements are well enough understood or standardised to be taken over without being re-created;
- trade-offs.

These are the contexts in which you can discover all of the elements described in Part I. These two parts of the book therefore form a matrix of elements against contexts. Your project will follow its own unique path through this matrix of discovery opportunities.

The book ends with a chapter on how to put all the elements and contexts together. Among other things, it describes the use of the element/context matrix itself to guide the planning of the requirements process.

1.9 Further Reading

1.9.1 Books on ‘Softer’ Approaches

1. Beyer, H. and Holtzblatt, K. (1998) *Contextual Design: Defining Customer-Centred Systems*, London: Morgan Kaufmann.

A good practical book from industry experts with a user interface design background. They describe a process that moves from the ‘customer’ to understanding the work context of a future product, to designing and prototyping a software product and its user interface.

2. Checkland, P. and Scholes, J. (1990) *Soft Systems Methodology in Action*, Chichester: John Wiley & Sons, Ltd.

Checkland essentially founded the Soft Systems Methodology (SSM) approach, and his books present the thinking and experience behind it. Checkland’s ‘rich pictures’ and way of thinking are useful for understanding a problem and the many pressures on systems and their stakeholders.

SSM has now taken on a life of its own, but Checkland explains the basic concepts well. SSM is a valuable precursor to requirements work, for example when people don't agree on what problem is to be solved.

3. Dewar, J. (2002) *Assumption-Based Planning*, Cambridge: Cambridge University Press.

Dewar's book describes a set of effective techniques to explore and improve strategic plans. It shows how you can discover the unspoken assumptions that plans depend on, and work out what to do if those assumptions should break. This leads to the reasons for decisions, and to robust requirements.

4. Heron, J. (1996) *Co-operative Inquiry: Research into the Human Condition*, London: Sage.

Heron has written a rather intellectual and academic book, but it describes a practical way for people (groups of stakeholders) to work together in an inquiry cycle process for any purpose. The approach can be seen as a human-centred version of the widely used 'plan, do, check' management cycle proposed by W. Edwards Deming. It is also a kind of 'action research'. A similar approach was used by Peter Checkland and his colleagues to develop Soft Systems Methodology.

5. Kotonya, G. and Sommerville, I. (1998) *Requirements Engineering, Processes and Techniques*, Chichester: John Wiley & Sons, Ltd.

This useful textbook pioneered the coverage of 'early' requirements work. Its authors who, like Checkland, were researchers at Lancaster University, were aware of that university's tradition of collaborative work including ethnographic observation and soft systems.

6. Goldsmith, R.F. (2004) *Discovering REAL Business Requirements for Software Project Success*, Boston: Artech House.

This is one of the few books explicitly about requirements discovery (and is very different from this book). Goldsmith offers many practical suggestions for checking and evaluating proposed requirements so as to weed out any that are not 'REAL Business Requirements'. The book is simply and engagingly written. It is very clear on some of the common traps and pitfalls in writing requirements.

1.9.2 Books on the Philosophical Background

7. Pinker, S. (2007) *The Stuff of Thought*, London: Allen Lane.

Pinker's wonderful book explains the structures and concepts that seem to lie below the surface of all human languages. What is fascinating is that many of the elements that go to make up requirements (as in Part I of *Discovering Requirements*) seem to be inherent in every language.

For example, sentences have place-holders for (stakeholder) roles in the form of expected subjects and objects; roles are expected to have goals, and to push for those, against opposing forces (threats, hostile stakeholders); special verbs describe (interface) events and have implied (scenario) timelines associated with them, and so on. If Pinker is right, then the described requirement elements are fundamental to human thought, and will always be the natural way to express needs.

8. Cohn, M. (2004) *User Stories Applied: For Agile Software Development*, Boston: Addison-Wesley.

Cohn has written a radical, funny and powerfully argued book. It effectively demolishes the traditional approach of, say, IEEE 830 (Institute of Electrical and Electronics Engineers Software Requirements Specifications), that requirements can be written as a straightforward set of 'shall statements'. Cohn shows that more is needed. He favours short, informal scenarios (user stories) but is wise enough to see that other elements, such as goals, have their uses (e.g. page 135 of his book). The approach as written is strongly tailored to software, e.g. with very rapid build cycles, but many of the ideas have wider application.

9. Gause, D.C. and Weinberg, G.M. (1989) *Exploring Requirements: Quality Before Design*, New York: Dorset House.

This was one of the very first books that was explicitly about 'requirements'. It remains stimulating and engaging all these years after it was published because it looks, very simply, at the basic issues that underlie requirements discovery, even if it raises many more questions than it answers. Here is a taste: 'In one requirements review of a single eight-page piece of an on-line banking system, we turned up 121 ambiguities that were interpreted in at least two ways by different reviewers.'

1.9.3 Books on 'Harder' Approaches

10. Simon, H.A. (1996) *The Sciences of the Artificial*, 3rd edition, Cambridge, Mass: MIT Press.

This small paperback contains a set of essays; Simon describes it as a 'fugue, whose subject and countersubject were first uttered in lectures ... but are now woven together as ... alternating chapters'. In other words, it is an intellectual account of the underpinnings of the hard, rational, systems view of the world of both engineering and society. Chapters 5 and 8, on design and complexity respectively, are the most obviously relevant. Simon writes fluently and persuasively, but from an extremely 'hard' standpoint.

11. Stevens, R., Brook, P., Jackson, K. and Arnold, S. (1998) *Systems Engineering, Coping with Complexity*, London: Prentice Hall.

'The green book', as it is known by systems engineers, is a simple account of how a large systems project for a product such as an aircraft should be organised. (*The Sciences of the Artificial* is listed in its references.) The book starts with a set of requirements to be managed; it is consistent with the philosophy of requirements database tools such as Stevens' creation, DOORS, but focuses on processes rather than tool support.

12. Alexander, I. and Stevens, R. (2002) *Writing Better Requirements*, London: Addison-Wesley.

Alexander and Stevens provide simple advice on framing requirements in words, with a minimum of process.

CHAPTER TWO

Stakeholders

The two most important parts of a computing system are the users and their data, in that order.

Neville Holmes

Requirement Elements	Discovery Contexts	Priorities						
		Measurements						
		Definitions						
Rationale and Assumptions								
Qualities and Constraints								
Scenarios								
Context, Interfaces, Scope								
Goals								
Stakeholders	Introduction							
	From Individuals							
	From Groups							
	From Things							
	Trade-Offs							
	Putting it all Together							

Answering the questions:

- Who has a valid interest in this product or service?
 - How do you engage with and manage those people?
 - Which requirements come from which stakeholders?
- ... so you know who you need to deal with on the project.

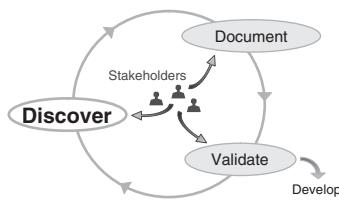
2.1 Summary

This chapter looks in turn at what stakeholders are, how to discover the stakeholders on your project, how to engage with them and how to manage them to ensure success.

Stakeholders are far more diverse than simply ‘developers’ and ‘users’. A better understanding of stakeholders as beneficiaries, operators, interfacing and negative stakeholders, regulators and others, contributes to more effective discovery of requirements.

The chapter ends with a look ahead (to the rest of the book) at how you will work with stakeholders to discover requirements of different kinds.

2.2 Discovering Stakeholders



Requirements ultimately begin and end with people – stakeholders. Constructing an onion model (Alexander 2005) [1] is a good way to start to understand and document the stakeholder structure of your project (Figure 2.1). Another starting point may be an organisation chart, as long as you don’t overlook stakeholders outside the organisation.

The rings of the ‘onion’ are centred on the product or service itself (as it is planned to be), not on the project and its team of developers.

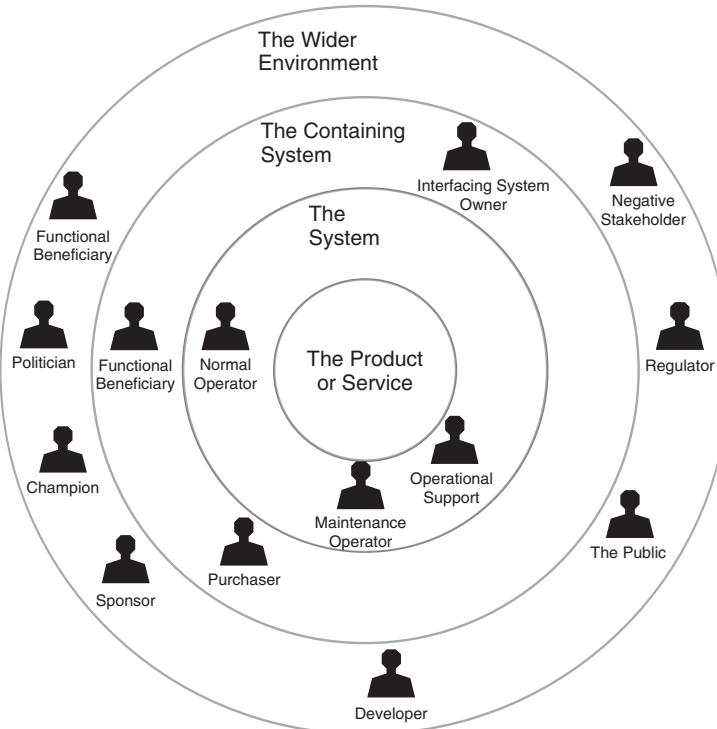


Figure 2.1: 'Onion' model of stakeholders in a typical system.

Developers: Inside or Outside the System?

Developers are very close to the product while it is under development but may have little to do with it once it is in service unless they 'wear two hats' by also having a specific operational role, such as maintenance or helpdesk (operational support). If the onion is drawn for the product when it is in service, the developer therefore appears in 'the wider environment'; the maintenance operator appears as part of 'the system'.

You could draw a *different* onion model for the product under development, in which case your system will be 'the development system' (your software factory, for instance). In that case, the developer will be inside, and operators outside. To a degree, we use onion models to counteract the tendency to marginalise operators and other stakeholders, so we tend not to focus too much on the development environment at this point.

Table 2.1: Operational roles within ‘the system’.

Generic role	Work done by role	Example
Normal operator	Interacts with the product to deliver results (to functional beneficiaries)	Tram driver drives the tram
Maintenance operator	Services and repairs the product (i.e. carries out both planned and unplanned maintenance)	Mechanic services the tram
Support operator	Provides help and co-ordination to keep the other operators productive	Roster co-ordinator allocates drivers to trams each day

2.2.1 Operational Stakeholders within ‘The System’

The innermost ring of the onion (around its solid centre) is ‘the system’. This means the people, procedures and products that together deliver results to the world outside. Notice that ‘the system’ is not the same as a product, a service, a computer or a piece of software, though it may contain any of those things.

The people who form part of the system around a product or service are the operational stakeholders: they take part in day-to-day operations.

Typical systems contain several operational roles¹ (Table 2.1). In a complex system, such as a warship or a manufacturing plant, there may be hundreds of different operational roles.

2.2.2 Stakeholders in the Containing System and Wider Environment

Beneficiaries: Who’s it for?

The man who pays the piper calls the tune.

Traditional English Proverb

Where the onion model in Figure 2.1 gives a product-centric view, Figure 2.2 offers an alternative, project-centric view, which may seem more familiar. This is because projects have traditionally thought of the world as consisting of two groups: themselves and ‘users’, by which they meant everybody else. Figure 2.2, which was created to help people in a transport organisation think about their stakeholders, goes a little further, distinguishing beneficiaries—people who are intended to benefit in some way—from the others.

¹You may hear the term ‘actors’, which is what the Unified Modeling Language (UML) calls operational roles. Note that if there is an ‘intelligent’ software system at the far end of an interface, UML considers that an actor as well. Essentially, UML is interested only in actively interacting roles, not in stakeholders in general. Interfaces are described in Chapter 4.

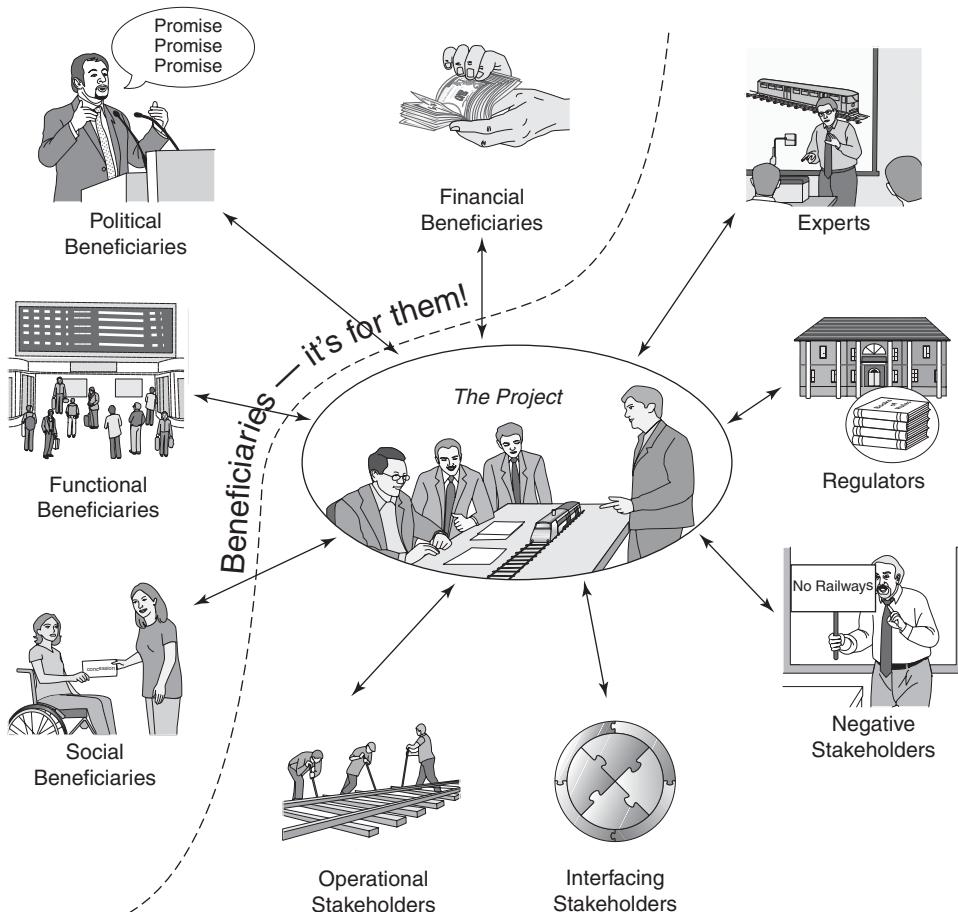


Figure 2.2: Beneficiaries and other stakeholders in a transport project.

A system, such as a tram, for example, forms part of some larger system or context. Systems are created to provide services or results to their beneficiaries. Beneficiaries include several very different kinds of stakeholder, but by no means all of them.

For example, the driver of a tram is not driving it for his own enjoyment. The intended beneficiaries of the tram service are the passengers, who pay to be carried along the tram's route. In fact, the service is for them.

You can answer the question 'Why are we operating this service?' by studying the benefits that it provides. Those benefits are not only to the passengers: the tram company, its directors and its shareholders benefit financially (as of course does the driver, in a small way).

We can call the passengers 'functional beneficiaries', and the shareholders (who are arguably further from the product, i.e. in the 'wider environment' ring), 'financial beneficiaries'. Projects also often bring political gains to people

whose careers benefit from a project's success. Such people may be career politicians, or may be political with a small 'p', within the corridors of your organisation.

Regulators

One role that is of crucial importance to services or products for the public is that of the regulator. There are regulators of financial correctness, of food, of medicines, of the radio spectrum, of broadcasting, of aviation, of railways, of manufacturing - in fact, of nearly everything.

Many products are subject to multiple regulators. Software is in nearly everything, so it is regulated in multiple ways (see box, 'Statutes, Regulations, Standards').

Statutes, Regulations, Standards

In some countries, such as the UK, regulation takes several forms:

- primary legislation in the form of **statute laws** passed by Parliament: for instance, the Financial Services Authority (FSA) was created by statute law;
- secondary legislation in the form of official **regulations** issued by government departments or other regulatory bodies, such as the FSA, under authority given to them by Parliament; these are often much closer than statutes to the level at which projects work;
- **international standards** imposed by bodies such as the International Standards Organisation (ISO);
- **national standards** imposed by general standards bodies such as the British Standards Institution (BSI), or specialist professional bodies such as the Institution of Engineering and Technology (IET);
- **industry 'best practice'** and '**guidance**', which are often essentially mandatory within an industry, especially where safety is concerned; businesses may comply with these voluntarily, to give their offerings a mark of quality and hence seek a competitive advantage, or may be obliged to comply by regulation.

As if all that were not enough, companies often impose their own **company standards** on their projects, e.g. imposing a software development process.

Projects, in turn, often write their own **procedures**, for example, describing how to structure requirements and use cases in their chosen requirements tool.

Standards and regulations can be seen as **reusable sets of requirements** that are shared between all systems in a domain. Some are voluntary ‘best practice’ or ‘guidance’ (see box, ‘Statutes, Regulations, Standards’); some are essentially mandatory. More is said on requirements reuse in Chapter 13.

Standards and regulations form a significant percentage of all requirements. This shows, incidentally, that not all requirements come from ‘users’.

Interfacing Roles

Almost every system has significant interfaces to other systems, services or businesses (whether existing or future). For example, many devices such as laptop computers and mobile phones have a Bluetooth² short-range radio interface, allowing them to exchange data with other Bluetooth-compatible devices.

Chapter 4, which looks at context, interfaces and scope, describes how you can analyse interfaces to identify requirements on your system. Effectively, your freedom of design is constrained by your interfaces, especially if they are already fully specified.

Identifying the systems or services that you need to interface to is obviously a vital step towards creating the right requirements and the right product. The project or organisation responsible for the other end of each interface is an important stakeholder in your project.

Negative Stakeholders

Generally, your project should aim to satisfy most of your stakeholders – but that does not necessarily include negative stakeholders.

Negative stakeholders range in attitude from peaceful opposition to active hostility. They may threaten or cause harm to a project, intentionally or incidentally. Your competitors may not wish your project to succeed, but they will generally stay within the law in their opposition to it.

Peaceful and lawful opposition can include, for example, writing letters to the government and collecting signatures to oppose the construction of a road, airport, shopping centre or power station that threatens to destroy assets valued by the stakeholders. Such assets can be of any kind, for example, historic buildings, the natural environment and wildlife, and existing jobs and businesses, as well as more abstract things such as leisure, safety, peace and quiet, and privacy. As a result, negative reactions to a project can be extremely diverse.

Security threats also come in many shapes: from thieves, vandals, hackers, disgruntled employees, criminal gangs using viruses and malware, terrorists

²Bluetooth is currently an industry standard under the control of a Special Interest Group (SIG), a trade association. Many industry standards eventually become International Standards.

and military enemies. Thieves and hackers may be opportunistic, rather than intentionally hostile. The others are, presumably hostile by intention.

For military systems, the enemy is a key stakeholder. He may use any means to confuse, disrupt, deceive or destroy. Countering these threats leads to many of the requirements on military equipment, for qualities such as data integrity, confidentiality and survivability (see Chapter 6, Qualities and Constraints). New threats, such as electronic and cyber-warfare, are increasingly important concerns.

You might imagine that software projects do not face the risk of opposition. However, the public are becoming concerned about data protection for reasons of privacy and financial security. Online advertising techniques such as stealthy data collection have recently created a storm of protest on the web. Viruses are able to attack smartphones and PDAs as well as computers. Negative stakeholders for software can include the public, bloggers, campaign groups, hackers, virus writers and probably many others.

Sponsor and Champion

Without a sponsor your project will be shortlived.

Suzanne Robertson [2]

In both commercial companies and public service, money is generally in shorter supply than ideas for how to spend it. Therefore, departments usually compete for resources. Any project that receives funding may be watched jealously for signs of weakness so that it may be killed off and its funds appropriated for something else.

The Champion has to be someone with enough power in the organisation to protect the project against 'political' threats (i.e. from negative stakeholders within the organisation). Mere technical success is no guarantee of safety. An effective Champion does not have to be technical, and is typically not involved in the day-to-day running of the project.

Sponsor and Champion

- The sponsor provides development funding for a project.
- The champion provides 'political' support for a project.
- The two roles can be combined but are often separate.

The Champion could be the department head who sponsors the project, but it's equally likely that the sponsor is another organisation or another part of

the development organisation. For example, an automobile maker could have a new secret project to develop a fuel cell car for the luxury urban market. This might be sponsored by international marketing as part of their roadmap for a future product line, but championed by the director of technology in board meetings.

Worked Example: Stakeholders in a Tram Service

The agency planning a new tram service must carefully consider possible opposition from residents, pressure groups (such as wildlife organisations), businesses and local government. All of these could be either beneficiaries or negative stakeholders. The direction they choose depends both on their own attitudes and on the preparation, skill and tact of the project's management, including its handling of requirements.

Figure 2.3 illustrates a typical geographic situation for a tram project. The City Tram is to relieve traffic congestion by providing a reduced journey time from A to B. With closely spaced stops at places such as C, D, E, F, G, H, and an interchange with the railway at B, it should encourage economic growth in the city. A stop at D provides access to public transport for residents of the district north of the park, and could encourage further expansion of the tram service to the north.

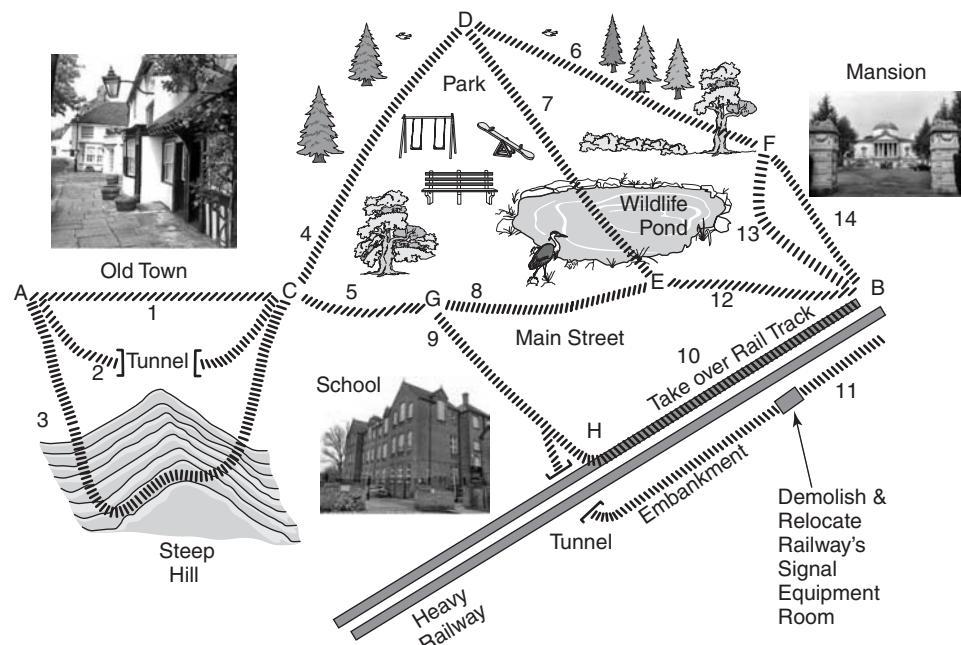


Figure 2.3: Tram routes: a problem in stakeholder geography.

The tram is likely to be supported by local businesses and by the developers of a new shopping mall on Main Street. Bus and heavy rail are also likely to provide support, as long as the interfaces to those services are well thought out.

However, there are obstacles. Between A and C is the old town, with an attractive townscape. A route through here could upset the town council (local government), who are otherwise likely to be in favour because of the commercial benefits a tram could bring. Local governments are powerful stakeholders, with authority over local planning issues.

To the south is a steep hill, which could cause operational difficulties for the tram in wet or icy weather. A central route is imaginable but would require a costly tunnel through the hill. Apart from the cost, a tunnel makes escape in case of fire much more difficult, so another powerful stakeholder, the railway safety authority, could cause difficulties here.

Between C and F is the town park, with scenic views, much-loved mature trees and a wildlife pond. Near F is The Mansion, a Grade I historic building protected both by law and by local and national conservation bodies; all are powerful stakeholders.

If the tram was to exit the park at E, it would avoid The Mansion but harm the wildlife pond. A partial alternative is to route the tram in a longer semicircular arc between F and B so as to avoid the street immediately in front of The Mansion.

The park could be avoided by a route through G, south of the park, but the route from G to E is along busy Main Street, threatening to worsen the traffic congestion rather than alleviating it.

An alternative to Main Street is the route southeast from G to the heavy railway at H; unfortunately, this crosses the school playground and could cause strong local opposition to the tram. From H to B, the tram could run on one of the existing railway tracks if the railway company can be persuaded to give it up, or it could run on top of the embankment south of the railway line. However, the embankment route would demand a tunnel to cross the railway, as well as the demolition and re-siting of the railway's signal equipment room, which the railway company opposes. The tunnel would raise buildability issues, as access to the railway is limited, and closures would have to be planned for nights and weekends.

The problem facing the tram project team, therefore, is to choose the best route (set of route segments linking A and B) by trading-off the constraints just described. You can see at once that each group of stakeholders (the Historic Building Society, the school's parent teacher association, the Old Town Preservation Society, the Wildlife Trust, the railway, etc) has an interest in at least one route segment, and could easily become actively hostile to the project. This kind of project has very many stakeholders (it is common to find hundreds of interested parties) and their management is a large part of the work of a successful transport project.

2.3 Identifying Stakeholders

There are several ways of finding out who your stakeholders are, i.e. who should be involved or consulted on your project:

- by asking your sponsor or client;
- by examining what is and what isn't shown on an organisation chart;
- with a template such as the 'onion model';
- by comparison with similar projects;
- by analysing the context of the project.

Naturally, as interviewing proceeds, you can always ask people: 'And who else should we talk to about that?'

2.3.1 From your Sponsor or Client

A natural place to start is your sponsor. Ask them who the other stakeholders are. Meet those people and ask them the same question, and so on. This would be an ideal approach but for one thing: project people may only lead you to people they know, within a tight group. This, is perhaps, a particular danger within a large organisation, where roles outside may be overlooked.

2.3.2 With a Template such as the Onion Model

The onion model or an equivalent list of typical project stakeholders can help to counteract closed-group thinking.

If you think graphically, take a copy of an onion model like the one shown in Figure 2.1.

- For each icon on the diagram, ask: 'Does our project have stakeholders with this role?'
- If the answer is yes, label that icon with the name of the role; be specific where the basic model is generic.
- Then, for each role you have identified, ask: 'Who interacts with this role?' or 'Who has an influence on this role?'
- Add new icons if necessary, and label them with their roles.
- Draw arrows between the role icons, and label these for each interaction or influence.

If you find it more helpful to use text, use the stakeholder roles listed in Figure 2.4 to identify people you need to contact, and follow this up with action.

Figure 2.4 shows a sample of typical, generic roles. Your organisation's classification may differ, as roles overlap, are lumped together or are further fragmented. In your area, people may use different names for some roles. Individual stakeholders may hold two or more roles, sometimes in surprising combinations. Expect to find a special situation on every project.

Operational roles
<ul style="list-style-type: none"> • Normal operations • Maintenance • Support • Helpdesk • Training • Planning, scheduling • Control, management, administration
Beneficiaries
<ul style="list-style-type: none"> • Functional beneficiary • Social beneficiary, etc (if indirect benefits exist) • Financial beneficiary • Political beneficiary (not only professional politicians)
Interfacing roles
<ul style="list-style-type: none"> • Owner of interfacing system (sharing data, etc) • Neighbouring business (mutual benefit) <p>Interoperating service (sharing facilities/equipment)</p>
Surrogate roles (representing or working on behalf of others)
<ul style="list-style-type: none"> • Champion • Purchaser (roles here vary widely and often overlap) <ul style="list-style-type: none"> - Internal customer - Procurement - Project sponsor - Marketing (representing the consumer) - Product manager • Developer (many roles possible here) <ul style="list-style-type: none"> - Requirements analyst - Designer - User interface designer - Programmer - Tester (very useful in requirements work) - Technical writer • Manufacturer/subcontractor/supplier • Expert/consultant (many more roles possible here) <ul style="list-style-type: none"> - Human factors/ergonomics - Safety engineer - Security engineer - Simulation/modelling expert - Legal opinion - Translator, cultural advisor, etc • Regulator <ul style="list-style-type: none"> - Parliament, statute law - Government departments, regulations - Statutory regulators - Standards bodies (national and international) - Voluntary/industry regulators

Figure 2.4: Template: possible stakeholder roles on projects.

Hybrid roles

- User (= Normal Operator + Functional Beneficiary)
- Consumer (= Mass-market Purchaser + User)
- Service Provider (= Operational Support + Maintenance + Helpdesk, and sometimes Developer, Subcontractor, Manufacturer)
- Risk and Revenue-Sharing Partner (= Developer + Manufacturer + Financial Beneficiary), etc

Negative/hostile roles

- Vandal, graffiti artist
- Thief
- Hacker, virus writer
- Competitor
- Industrial espionage (via malware, etc)
- Fraudster
- Disgruntled employee
- Trades union
- Political opponent
- The public, residents' association, etc
- Activist, environmental pressure group, etc
- Military enemy, terrorist

Figure 2.4: (Continued)**Surrogate Roles**

You will find stakeholder roles much easier to unpick once you understand one common situation: many roles, including every kind of paid work, are surrogates, i.e. on behalf of somebody else.

For example: a company director represents the shareholders; a politician represents the public; a lawyer represents a plaintiff; a requirements engineer represents a project's stakeholders; a user interface designer represents a product's human operators.

The breadth of this list of examples indicates that surrogacy is the usual case on development projects. The old dogma, 'Go and get the requirements from the users directly', is well-intentioned, but wrong for several reasons:

- There isn't one uniform group of people who use a product.
- You don't know what using the product will be like until you've designed it!
- 'The users don't know what they want until you show it to them.' Products create requirements, as much as the other way round.
- You often can't talk to users directly but you must deal with surrogates, including yourself and your project colleagues.

Surrogacy is valuable because it enables a project to deal with a manageable number of people, and because it means that external parties are represented. This is necessary for several reasons:

- Large and complex projects can take many years, making it impossible to talk to ‘the users’. If the aircraft carrier that is being planned today will not sail for another 25 years, many of its crew will not yet have been born. The future military and political environment cannot be fully known either. Planners and modellers are expert in working out the range of likely future ‘worlds’.
- Government and its appointed regulators represent the law and the public – the voice of the people. It isn’t possible to speak to millions of people, but the officially-appointed regulator is empowered to speak for them.
- A mass-market product like a car or music player will have millions of users. Their opinions can be surveyed, or a sample can be involved (e.g. to comment on a prototype). Marketing and product management professionals have specific expertise in identifying and predicting what the consuming public want.

Surrogacy is dangerous because people can be wrong about the needs and wishes of the people they claim to represent. (That is why old timers tell you to go and talk to ‘the users’.)

- Marketing and product management have to live with conflicting interests; they represent the views of the market to the company, but also serve the company.

The best surrogacy efficiently condenses many vague feelings and intentions into a clear, representative statement. You will have to use surrogates on your project – you are one yourself. Your task in discovering other people’s requirements is to serve those people as faithfully as possible. A measure of scepticism and self-knowledge is needed.

2.3.3 By Comparison with Similar Projects

If you have access to people who have experience of similar projects, ask them for their list of stakeholders, or get them to look for gaps in your list.

2.3.4 By Analysing Context

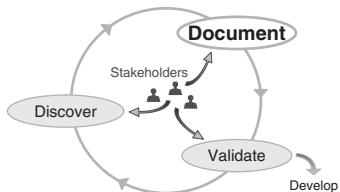
Another approach is to explore the context (see Chapter 4) of your project. For example, if your context is geographical (as with the tram service), then

searching a map of the area immediately around your tram route may reveal neighbouring businesses and services. Or again, if your context is an existing technical or service domain (e.g. hospitals), then analysing the roles and responsibilities in that domain (nurses, physicians, radiology . . .) is likely to be productive.

Tips for Discovering Stakeholders

- Listen for names of people and organisations when you are starting on a project.
- Find out about the roles and organisations that people mention.
- Sketch an onion model in your notebook; add names and organisations to it; draw in relationships when you discover them.
- Use the template (Figure 2.4) to help identify missed roles.
- Once you have started analysing context, check your context knowledge against your stakeholder list.

2.4 Managing Your Stakeholders



Answering the question:

If there are so many people playing so many different roles in the project, how do we manage them all?

2.4.1 Engaging with Stakeholders

People are usually happy to have their views and opinions heard (see Chapters 11 and 12). The key to engaging with people on a project is simply to be open, honest and friendly, not to take sides, and to follow up each person's questions or suggestions appropriately. It also means taking great care with people's feelings. You need to keep track of your dealings with stakeholders, to analyse influences on your project, and to prioritise your activities accordingly. Managing stakeholders is more than discovering and documenting them – it's an ongoing negotiation.

Table 2.2: Basic stakeholder management list.

Role	Name/ organisation	Contact details	Actions to be taken by project	Agreement status	Associated issues
Interfacing	Old Town Bus Company (OTBC)	Jane Brown (Business Planning) 151 Main Street Tel 0123–4567	Negotiate position of bus/tram interchange near School Street	Preliminary; contact made	Effect of tram on bus passengers (increase or decrease)
Political	Anytown Council	Alderman Joe Bloggs (Transport Planning) The Town Hall 1 The Square Tel 0123–4121	Agree outline approach	Council likely to give permission	Elections in September
...

2.4.2 Keeping Track of Stakeholders

Stakeholders are the people who have influence on your project. That means you need to keep them happy. So, it's vital that you pay attention to them. At the very least, you need to know who they are, what they want, and whether they have agreed to the requirements. Table 2.2 illustrates a simple way of keeping track of your stakeholders.

Keep in mind that each stakeholder group (and individual stakeholders within groups) have different expectations. An important strategy is to keep stakeholders informed as the project proceeds. It is rarely possible to meet everyone's expectations. By providing periodic project status updates, you can manage people's expectations and improve the likelihood of project success.

2.4.3 Analysing Influences

Figure 2.5 illustrates a simple analysis of influences for a video game development project. Influences are drawn as an overlay of labelled arrows between stakeholders on an onion model.

This is an informal diagram, and the only rules are to draw whatever is useful for your project. (A mathematical analysis of influences is possible, but is unlikely to be either feasible or cost effective on industrial projects, as it requires extensive data collection.) In the diagram, a different colour or style of line is used for each kind of influence.

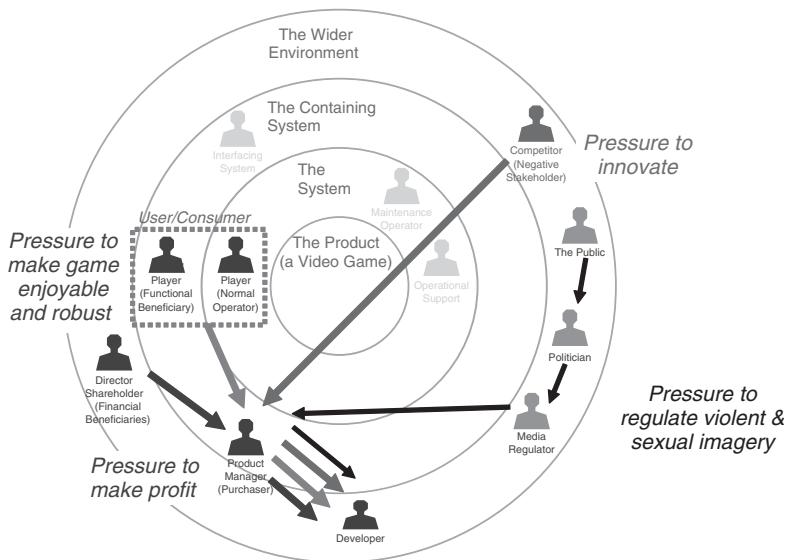


Figure 2.5: Analysing stakeholder influences on product development.

There are strong competing pressures on the maker of a video game:

- The consumer audience is savvy and aware of the market, and shares opinions freely on the Internet. In other words, reputations can be made or broken rapidly. Game players want new games to be exciting, scary, varied, emotionally intense, involving, realistic and action-packed. This all means development work and cost, on artistic design, music, game structure, interactions and so on.
- Competitors exert strong pressure to innovate or perish; the short period during which a game is new is its only opportunity to sell profitably.
- The public, through pressure on politicians, create media regulation of violent and offensive imagery.
- Meanwhile, the company's financial beneficiaries demand profit.
- The product manager, who is effectively the purchaser (the internal customer, representing the mass-market consumers), is responsible for integrating all these conflicting requirements into an effective specification, and transmitting these to the developers. This is requirements creation at the sharp end.

2.4.4 Prioritising Stakeholders

Projects often prioritise their stakeholder management activities by the relative power of the stakeholders. This is pragmatic, as long as you do not neglect less powerful stakeholders. One tried and tested approach is to make a matrix

of stakeholders against the kind of influence they have on the project: whether positive or negative, powerful or weak (politically), as in Table 2.3.

A Consumer is Not a Typical Stakeholder

Figure 2.5 illustrates the roles of ‘user’ or ‘consumer’. These are not basic stakeholder roles like operator, regulator or functional beneficiary at all.

Instead, a player both operates a video game and benefits from it functionally (by enjoying the game). This hybrid role is called ‘user’. Where the same person purchases the (mass-market) product, the role is a triple hybrid. Sometimes there is also a maintenance role involved.

Such all-in-one roles are not typical of stakeholder roles in general. Your personal experience of being a mass-market product consumer is not a good guide to understanding stakeholders in general.

As with any kind of risk register, the influence matrix is only of value if someone uses it to manage the project. On a project where influences matter, you should have a stakeholder manager who ensures that the requirements are understood and agreed, at least by the most powerful stakeholders. This role is closely related both to project management and to requirements management, so the task cannot be done in isolation.

Table 2.3: Influence matrix.

Stakeholder	Negative		Positive	
	Strong	Weak	Weak	Strong
Old Town Residents' Association		At the moment, the OTRA is undecided. With careful integration, could become positive.		
Anytown Council				Currently very keen. Could change at local council elections in May.
Mansion Preservation Society		Opposed to any route within 100m of Mansion; otherwise in favour.		
...				

2.4.5 Involving Stakeholders

You may need to choose your development approach to ensure that stakeholders are sufficiently involved in your project.

For example, you could choose a staged approach in which you carry out a feasibility study and obtain detailed feedback from stakeholders. You may then build a demonstrator, again obtaining feedback, before launching the main development process. The choice of life cycle depends on the nature of your project, the technological risk it entails and the stakeholder situation.

If you know of any negative or hostile stakeholder groups, you have an especially urgent task to make peace with them, or at least to defuse their negative feelings. Find out what they are concerned about; make sure they are properly informed, and set right any misconceptions they may have; and show that the project is professional and fair. If at all possible, take time to negotiate an agreed way forward with them.

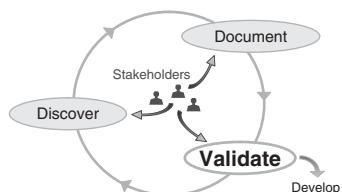
2.4.6 The Integrated Project Team

Where you are developing a product for use by a specific group of people – say, a healthcare team, engineers in the army or air traffic controllers – then the product is far more likely to be accepted if some representatives of that group form part of the development team.

An Integrated Project Team (IPT) includes people with all the skills and knowledge needed for the project to succeed. That may mean business analysts familiar with the domain, designers, programmers, human factors specialists, testers, and representatives of all the operational roles working with the product.

Creating an IPT sounds costly, but it need not be. IPT members do not have to be full time, as long as they are available often enough for communication to be easy and effective. Having operational stakeholders around to comment on a document or prototype as soon as it is made, and to suggest small changes that can be implemented in a few minutes, is far better (and, in the long run, cheaper) than relying on document deliveries and formal review comments.

2.5 Validating Your List of Stakeholders



Stakeholder groups are often fluid. People change their minds; companies are formed, merged or taken over; pressure groups become active and disappear.

For instance, in the context of the tram example, during an economic boom, property developers are very keen on new civic schemes such as shopping centres, and consequently (or in return for permission to build) they may be active supporters of transport schemes; they rapidly disappear, however, when times become harder.

Expect, therefore, to have to check and update your stakeholder knowledge regularly to keep your list ‘complete’:

- Make it a regular action on your project to check the stakeholder list.
- Regularly review actions on the stakeholder list and ensure they are taken.
- Update the stakeholder list to include people and organisations who have recently communicated with the project or who have been mentioned in project reports.
- Use the influence matrix to track changes to stakeholder positions.

2.5.1 Things To Check the Stakeholder Analysis Against

- Neighbours mentioned in the context model (Chapter 4).
- People mentioned in interviews (Chapter 11).

2.6 The Bare Minimum of Stakeholder Analysis

- Identify who could ‘stop the show’ on your project.
- Find out what those people want, and negotiate as necessary.

2.7 Next Steps: Requirements from Stakeholders

Answering the questions:

- OK, so we know who our stakeholders are: how does that help us?
- What do we do next?

Your stakeholders will lead you to many of your requirements, other than those you can discover for yourself by, for example, the analysis of existing systems or mathematical modelling.

How you’ll create the requirements is the subject of the rest of this book. In general terms, Table 2.4 shows which stakeholders you will be most likely to work with to create each kind of requirement.

Table 2.4: Operational roles within 'the system'.

Stakeholder role	Type of requirement	Discovery technique
Normal operators (possibly in many different roles)	Scenarios (Chapter 5) Usability (Chapter 6)	
Interfacing	Interface definitions (Chapter 4)	Interview (Chapter 10) Apprenticing (Chapter 10) Observation (Chapter 10)
Maintenance	Maintenance functions/scenarios (Chapter 5) Diagnostics, built-in test	Workshops (Chapter 11) Data modelling (Chapter 8) Prototyping (Chapter 12) Archaeology (Chapter 12)
Support	Support functions	
Functional beneficiary	Product functions/scenarios (Chapter 5) Performance targets (Chapter 9)	
Financial beneficiary	Mission, objective (Chapter 3)	Interview (Chapter 10) Read policy documents
Regulator	Regulations, laws, standards, guidance Responses to safety case, compliance statements, etc	Legal advice on regulations, etc Negotiate compliance
Experts, specialists in disciplines	Safety, security, reliability, usability, etc (Chapter 6) Constraints (Chapter 6)	Analysis, simulation, modelling, standards
Manufacturer	Producibility	Interview (Chapter 10) Workshop (Chapter 11) Prototyping (Chapter 12)
Marketing (surrogate, on behalf of mass-market customers)	Mass market (consumer) Preferences by group (age, income, etc)	Market survey, Field trials Observation (Chapter 10) Prototyping (Chapter 12) Analogous products (Chapter 12) Competitor analysis
Product manager, purchaser	Priorities Programme, schedule Budget (cost)	Prioritisation (Chapter 13) Trade-offs (Chapter 14)
The public	(Lack of negative impact)	Public meetings, Focus groups, Consultation, Roadshows

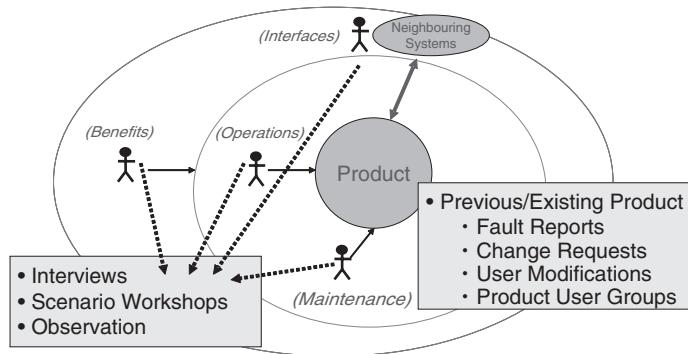


Figure 2.6: Requirements from operational stakeholders.

You will use a range of techniques described in Part I, such as goal and scenario modelling (see Chapters 3 and 5), to discover intended benefits and operational scenarios (Figure 2.6). You may also do some ‘product archaeology’ to discover possible requirements from fault reports and the like on the existing product.

You will similarly use a range of techniques and sources appropriate to your project to discover requirements with your non-operational stakeholders (Figure 2.7).

In each case, you should validate your findings with your stakeholders.

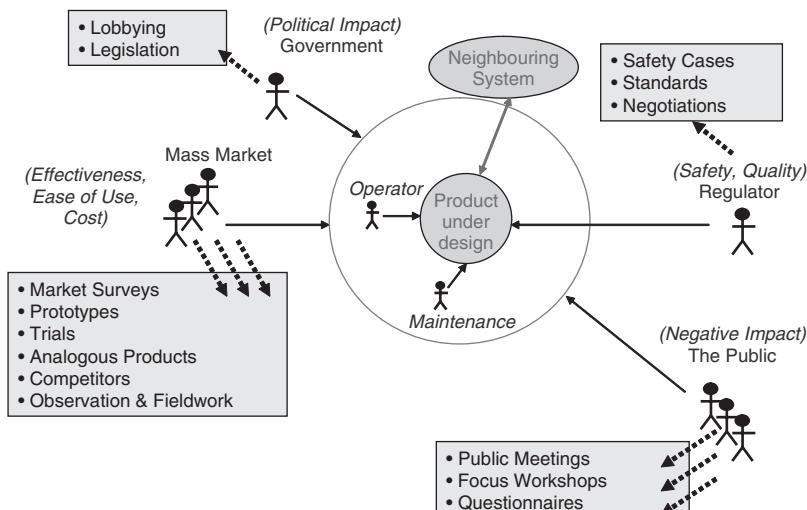


Figure 2.7: Requirements from non-operational stakeholders.

2.8 Exercise

You are a product manager for a machine tool company. The directors have asked you to develop a new cutting machine to cut cloth for fashionable dresses of all sizes and patterns. The machine will be sold to clothing makers around the world:

- a. Who are your key stakeholders?
- b. How will you analyse and validate your stakeholder list?

2.9 Further Reading

1. Alexander, I. (2005) A Taxonomy of Stakeholders: Human Roles in System Development, *International Journal of Technology and Human Interaction*, 1(1), 23–59.

Ian's paper on stakeholders is an academic analysis of stakeholder roles in a development project. It goes into more detail on each type of stakeholder than is possible here.

2. Robertson, S. and Robertson, J. (2004) *Requirements-Led Project Management: Discovering David's Slingshot*, Boston: Addison-Wesley.

Chapter 3, on project sociology, in the Robertsons' excellent book provides several different points of view on project stakeholders, including Belbin's team roles, Ian's onion model, and looking for the knowledge on different areas, such as security.

CHAPTER THREE

Goals

Goals are dreams with deadlines.

Diana Scharf-Hunt

Requirement Elements	Priorities					
Discovery Contexts	Measurements					
	Definitions					
Introduction	Rationale and Assumptions					
From Individuals	Qualities and Constraints					
From Groups	Scenarios					
From Things	Context, Interfaces, Scope					
Trade-Offs						
Putting it all Together						
Goals						
Stakeholders						

Answering the questions:

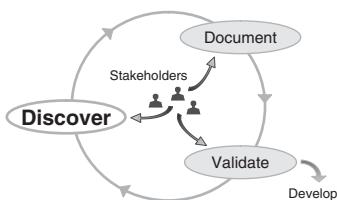
- What are you trying to achieve?
- What is this for?
 - ... so you know what the point of the project is.

3.1 Summary

A goal is something that a stakeholder wants to achieve. That may be to do something, or it may be to achieve or improve a quality such as reliability. Goals are your first description of stakeholder intentions: your first sketch of their requirements.

Goals are permitted to be neither fully achievable nor measurable, unlike requirements, which must be both. Goals may conflict, whereas requirements must not. Therefore, goals do not necessarily translate directly into requirements. Goals must be analysed and refined into realistic and measurable targets. Any conflicts must be resolved to create workable requirements.

3.2 Discovering Goals



Goals are things that (some) stakeholders want to achieve. Goals can be at any level of detail. The highest level and largest scale goals are mission statements and objectives. Long-lived business goals are called policies. The lowest level goals are individual functions. (For a discussion of different levels, see Appendix B.)

Detailed goals turn into requirements once they are:

- fully verifiable (see Chapter 9); a requirement is verifiable when you know how to test it or otherwise prove that it has been met;
- prioritised (see Chapter 10) by the project; a requirement is prioritised when you know how much it matters in a given system context. This is generally after trading-off candidate designs against goals (see Chapter 14).

Goals, Large and Small

People use different words for goals of different sizes. Some of the most often used are ‘mission’, ‘vision’ and ‘objective’.

Mission or Vision

The mission is the basic purpose of an organisation or project - the single thing it is set up to achieve. A clear mission is vital. The details can come later.

Perhaps the most famous mission statement of all was President Kennedy’s: ‘To put a man on the moon by 1970’. Whether that mission was wise, and what motivated it, can be debated. But it was clear and simple, and it drove the enormous Apollo project to a successful conclusion. Kennedy’s words enabled everybody to understand instantly what they were trying to do. That simplicity contributed powerfully to making the project coherent, single-minded and effective. Your project’s mission (or your *vision*) should be as clear.

Objectives

Once people have a definite mission for a business or a project, they can identify their objectives for it. An objective may be long term, but it should be measurable; you should know when you have achieved it.

Some people use ‘objective’ and ‘goal’ interchangeably. It is probably worth saving the word ‘objective’ for large, long-term goals. These are often imposed on a project by the business, i.e. the commercial or policy-making direction of a company.

Goals are not necessarily verifiable, and not necessarily agreed by all stakeholders. Making them so is a crucial part of the process of discovering requirements (Table 3.1). But you can’t do everything at once. You should start by identifying people’s goals, regardless of whether these are verifiable.

The goals of different stakeholders may well be in conflict with each other. That is normal for goals, but disastrous for requirements.

A project team can often determine the main goals, and hence the basic conflicts within a project, in a few minutes. These goals govern the shape of a project for the rest of its life. Stating them is enormously valuable. Identified goals and goal conflicts lend clarity to what is otherwise a continuous muddle.

- Projects that lack clear goals struggle constantly to understand what their real requirements are, and are unlikely to discover them.
- Projects without goals are vulnerable to pressure to add requirements, even if they don’t have the time or money for more work.

Table 3.1: The difference between goals and requirements.

(Stakeholder) goals	(Product) requirements
Belong to different stakeholders	Agreed by all
May conflict, indicating design trade-offs; these often drive project design activity and the choice of life cycle (e.g. iterations with analysis or prototypes of competing options, to reduce risk)	Must not conflict in the chosen technology; therefore, design envelope must be known to a sufficient degree (leaving as much freedom inside that envelope as possible)
May be an ideal, unattainable, indicating what is hoped for	Must be realisable within limits of budget, timescale, technology, and skill available

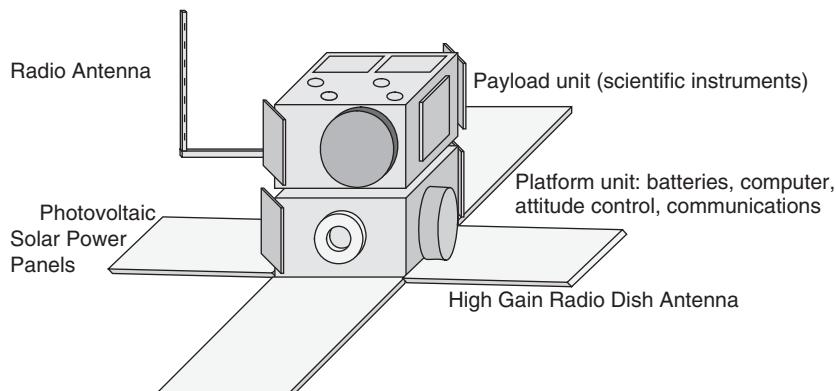
- Unstated conflicts lead to tension and confusion in the project.
- Stated but unresolved conflicts lead to useful action, such as making trade-offs and exploring design options.
- Having clear goals is a major factor in reducing rework.

In essence, goals are simple, practical, human things. They can work out very differently on different kinds of project, so we will give some detailed examples here to show how varied they can be.

3.2.1 Worked Example: Goals for a Spacecraft

Think about a scientific spacecraft (see Figure 3.1). Its highest-level goals are simple:

- to fly in space;
- to do worthwhile science;
- to send the results back to earth.

**Figure 3.1:** A typical scientific spacecraft.

Doing worthwhile science means carrying plenty of instruments, or perhaps a few clever and long-lived instruments that can discover plenty of new things.

The more instruments there are, the more they weigh, or rather (since we are in space), the larger their mass (in kilograms). Mass is very costly on a spacecraft; it may cost \$100,000 or more to put one kilogram into orbit.

Also, each instrument uses electrical power. Power is very difficult and costly to provide in a spacecraft:

- You can take batteries, but they will quickly be run down and they use up yet more mass.
- You can add solar cells to generate electricity, and recharge the batteries. They, too, take up mass. If they are big, they need a complicated mechanism to unfold them like wings. The spacecraft has to be launched with the wings folded. Once in space, they are unfolded. If the wings get stuck and refuse to unfold, the whole mission is ruined for lack of electrical power.
- You can add a radioactive source that emits heat as it decays. You can put that inside a generating machine that uses the heat to make electricity for many years (slowly getting less powerful). The radiation from the radioactive decay is dangerous to people while the spacecraft is on the ground. It would be even more dangerous if the spacecraft's launcher failed and the radioactive source crashed to earth. Even in space, it could damage delicate electronics in the spacecraft and its instruments. So, the source has to be mounted on the end of a long arm, which adds mass.

Flying in space means going in the desired direction or maintaining the desired position all the time, such as in orbit around the earth. That is achieved by firing small rocket motors to push the spacecraft back into the right position. That, in turn, means using up fuel, which has a mass.

Sending the results back to earth means transmitting and receiving radio messages. Radio transmitters and receivers have a mass, and use electrical power. The more data you want to send, and the further you are from earth, the more power you have to use. You can save some power by using bigger dish antennae, but that means more mass.

Staying in space for a long time means keeping every vital function working. It is usually impossible to repair a spacecraft if it fails. Space is a hostile place, as radiation damages electronic components such as computer memory chips. Vital functions can be protected by redundancy, i.e. having spare equipment. But that takes up more mass. It also makes the spacecraft more complicated, which makes it more costly and harder to test.

These trade-offs (see Chapter 14) suggest that the crucial quantities to be managed in a spacecraft are its mass and its power consumption. Almost

everything you want to do implies a demand for mass and a demand for power. The engineering problem is to trade-off the different goals to achieve a winning compromise and a workable all-up mass and power budget. With mass and power so limited, the instruments compete against each other for the right to fly.

An outline of the main goals for a spacecraft is shown in Figure 3.2, using a simple notation, with bubbles for goals, and arrows to show support or (if negative) obstruction. (See 3.3 Documenting Goals.)

Large abstract goals (like ‘do worthwhile space science’) quickly break down into concrete, usually functional, goals (like ‘generate 200 watts of electrical power’). These goals, in turn, break down into design choices, like ‘(delicate) solar cells in folding wings’ versus ‘(dangerous) radioactive source on a long arm with a thermal electric generator’.

The tricky choice of power sources emphasises another key goal for space engineering: reliability. When any problem may mean a disaster for the mission, you want to avoid or prevent problems. That makes space engineers prefer simple, well-known design choices. But doing new science means trying new designs for instruments, so there is an awkward trade-off there too. Qualities like reliability and safety are often key goals for projects on the earth as well as in space. That means possibly sacrificing functions or other qualities; naming a key goal implies the choice has been made to direct effort in that direction.

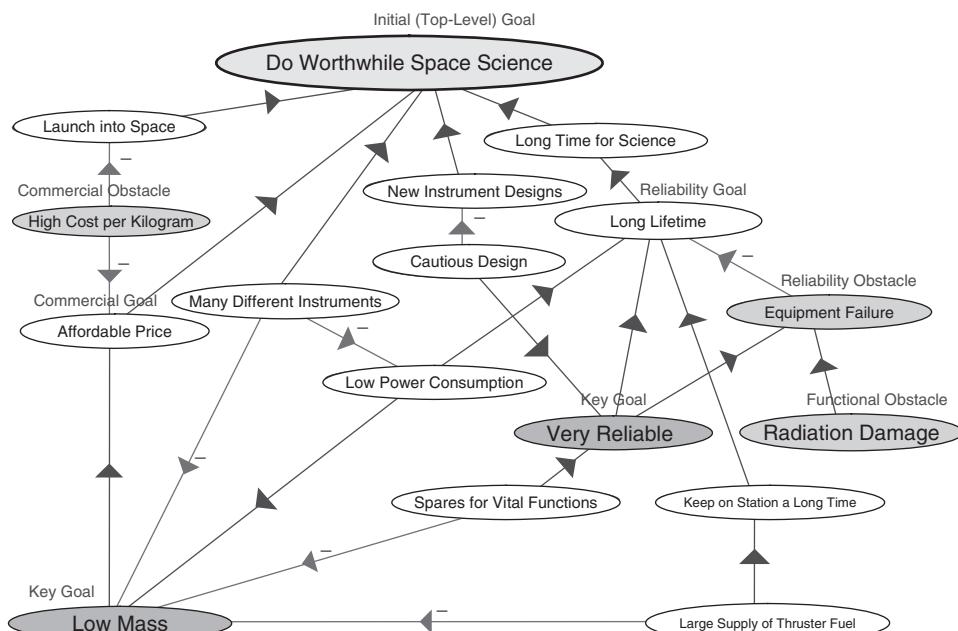


Figure 3.2: A spacecraft's design is severely constrained by a few key goals.

3.2.2 Worked Example: Goals for a Restaurant

For an example closer to earth, let us look at a restaurant project.

In a restaurant, different stakeholders have very different, even conflicting, goals. In such a situation, record each stakeholder's goals separately, so you know who owns each goal. You can do this on a flipchart or whiteboard by drawing a table with a column for each stakeholder, or simply by drawing a circle for each stakeholder and listing the goals inside (Figure 3.3).

The **owner/manager** wants to make a profit, and to have a business that is not too much trouble to run. This means, for example, having a good cook, efficient staff, and low maintenance and running costs.

In contrast, the **cook** wants good pay and to develop a reputation so he can have a successful career as a famous chef. That means he wants the restaurant to enable him to prepare delicious and distinctive food. He doesn't want any problems with hygiene or staff absence. That in turn means he needs top

Role	Cook	Owner/ manager	Customer/ diner	Supplier	Public health department	Taxman
Goals	Good pay Reputation No staff absences ...	Profit Top chef Low running costs ...	Good food Value for money ...	Paid on time, in full	Hygiene rules obeyed	Correct tax paid on time



Figure 3.3: Two ways of listing stakeholders' goals for a restaurant.

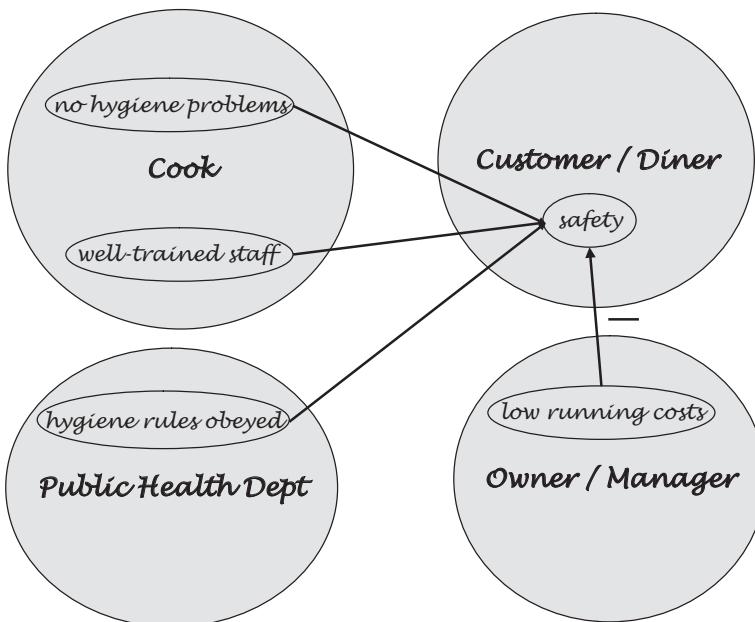


Figure 3.4: Support and conflict among restaurant stakeholders' goals.

equipment, well-trained staff, the best raw materials and, crucially, a large enough budget.

The **customers** who eat in the restaurant want good food and value for money; they want a predictable price, good service and an enjoyable ambience. They certainly don't want to get food poisoning, so safety is important to them.

The **suppliers** of food and other materials want to be paid, promptly and in full.

The **local council's public health department** want to see that hygiene rules are obeyed.

The **taxman** wants to know that the proper amounts of tax are being paid by the business.

Many of these goals support each other; a few of them conflict. For example, the public health department's 'hygiene rules obeyed' supports the customer/diner's 'safety', as does the cook's 'well-trained staff'. These could conflict with the owner/manager's goal of 'low running costs', however (Figure 3.4).

Functional and Quality Goals

A **functional goal** is to do something (e.g. 'to carry up to five passengers').

A **quality goal** is to change the way something is done (e.g. 'in comfort').

Qualities (see Chapter 6) include safety, security, reliability, performance, maintainability, total cost of ownership (purchase price + annual running costs), etc.

Top-level goals are often vague and unmeasurable qualities, such as 'safe (when travelling)'. These are not much use as requirements, but they often break down into several smaller, more precise qualities, such as 'accuracy of steering', 'safety of braking', and 'visibility to other drivers'.

These in turn often break down into smaller, directly measurable (and testable) functional goals and features, such as 'headlamps of 3,500 lumens', 'antilock braking', 'traction control', etc.

In general, customers often choose between products on reputation for qualities such as safety and reliability (e.g. they buy a Volvo car or Sony electronics). Such large qualities are too vague to test, however, so development teams work with smaller goals.

3.2.3 Worked Example: Tram Goals and Trade-offs

Let us continue the tram example from Figure 2.3 (Tram Routes) in Chapter 2.

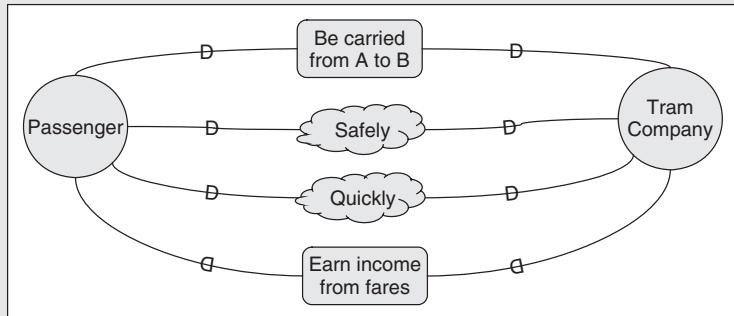
A tram service offers to provide fast, safe and reliable travel for passengers (and their shopping) around a town. It does that at the price of a considerable investment in infrastructure. That includes the trams themselves,

Analysing Goal Dependencies

Goals and stakeholders work together. The i* ('eye-star') goal modelling approach [4, 5], developed at the University of Toronto, extends the idea of grouping goals by stakeholder to looking at who depends on whom for what kind of service.

The i* notation covers two kinds of diagrams.

Strategic dependency diagrams show goals as relationships between pairs of stakeholders. Any number of **stakeholders** may be shown on a diagram. In the simple example here, a passenger **depends** (arrows marked 'D') on the tram company for being carried from A to B, a **functional goal**; and for that to be done safely and quickly: quality goals. i* calls such qualities '**soft goals**', and draws them as clouds.

An *i** strategic dependency diagram

Strategic rationale diagrams analyse the identified stakeholder **goals** in more detail, to derive **tasks** that can be implemented as software functions, etc. These are shown **inside the circle** drawn for the stakeholder. You can think of this as combining the simple notations of Figure 3.2 and Figure 3.4. So far, however, *i** has not been used much in industry, possibly because the notation seems complex - only a few of the *i** symbols are used here, and some have been modified here to make them easier to draw with standard tools.

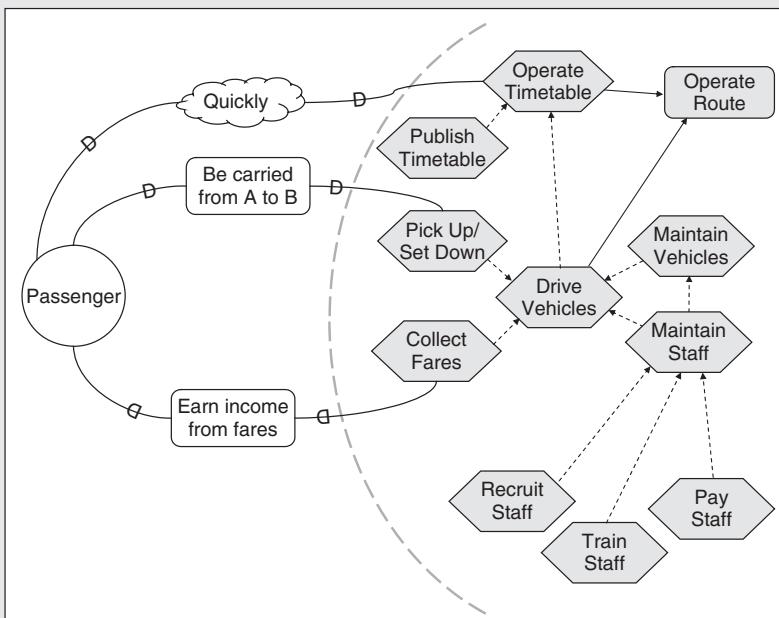
An *i** strategic rationale diagram



Figure 3.5: Trams and people in Milan (Italy).¹

street redesign (see the photograph Figure 3.5), track, tram-stops, information systems, ticketing, maintenance depots, and so on.

Introducing a tram causes side effects. One likely effect is disruption to other modes of transport. Car drivers may find sharing a road with a tram difficult. They may not be allowed to drive on the tram tracks. Some roads will be made one-way. It may be difficult to cross the tram route. The remaining routes may become congested by queues of cars.

Trams can have complex effects on people and businesses, both positive and negative. People can travel further and faster on some routes by using the tram. But they can take people away from businesses as easily as bring people in. A tram stop in front of a shop or a hotel could bring more customers by tram, or it might put off customers who like to arrive by car.

Each stakeholder has a different set of goals for the tram service (Table 3.2).

Depending on the local situation, many other stakeholders, such as youth and sports clubs, could be involved in a tram project. The tram problem shows that very complex networks of stakeholder goals and conflicts can build up from a simple proposal like 'let's build a tram from Old Town to White Mansion'.

¹The sign says 'Trams at walking pace'. Several compromises are visible in the photograph. The street is one-way, to accommodate the tram; the road is also shared, despite the safety implications, by people and motor traffic. Towns that do not already have such an infrastructure, and do not have a public that accepts trams on streets, face a daunting task.

Table 3.2: Stakeholders' goals for a tram service.

Stakeholders	Goals
Tram company	Make a profit on the service
Transport authority	Create successful transport schemes
Town council	Improve transport for local people (on whose behalf the council acts) Get more customers for local businesses Attract more businesses into the area Get increased revenue from businesses Get funding from developers
People in the town	Travel to work, shopping and leisure quickly, cheaply, reliably, and safely (whether by car, tram, bicycle or other mode of transport) Not have their homes blighted by noise and traffic
Developers	Make money from developing new shopping centres, etc, near to tram stops/interchanges with other modes of transport
Heritage, nature and school groups	Protect their resources from damage by the tram Reduce pollution
Local businesses	Get more customers (brought by the tram) Not to have shopfronts and deliveries obstructed (e.g. parking places lost) Not to have their customers taken away to other areas

3.2.4 Finding Solutions to Goal Conflicts

Individuals can hold conflicting goals, as illustrated in the tram case by both local people and businesses. In some situations, no workable resolution may be found. In the case of the tram, such situations should be predictable from an early stakeholder and goal analysis, but only after candidate routes (design options) have been identified. This is because the choice of solution (e.g. a tram route) changes the problem (which depends on the set of stakeholders affected by the route).

In general, it is only possible to decide whether a system is feasible and can meet its goals well enough, when sufficiently detailed design options exist.

- Sometimes, it is necessary to work out a candidate design in almost complete design detail.
- In other cases, a simple sketch will be enough to determine that an approach is not workable.

The tram example is continued in Chapter 14, with an exploration of the trade-offs between design options and stakeholders' goals.

Goals in Theory and Practice

In theory, theory and practice are the same.

In practice, they are different.

Yogi Berra, Baseball catcher/manager

The goals of local businesses show that there is a huge difference between a theoretical goal and its realisation in practice.

- A business will be in favour of the tram if it believes the tram will bring more customers.
- A business will be opposed if it believes the tram will make it harder for customers to come and shop with them.
- A business will be bitterly opposed if it thinks its customers will be taken away to a larger centre made newly accessible by the tram.

Thus, different businesses will be very interested in the predicted effects on traffic, parking, customer flows in and out of the area, and proposed locations of tram stops, and will support or oppose the project accordingly.

3.2.5 Contexts for Discovering Goals

To build up a picture of goals piece by piece, you can interview people individually (see Chapter 11). This favours attention to detail, possibly at the expense of the big picture.

The alternative is to hold a goal workshop, to bring together different stakeholders. Any disagreements over the big picture will emerge, and the tensions that drive the project will become clear. Chapter 12 gives general guidance on running workshops.

If you are using both interviews and workshops, you need to consider how to relate the two together. This can be done either by conducting interviews first or by running workshops first (Table 3.3).

We have used both approaches on different projects. Each has its merits; the choice is often driven by the client's preferences and timescales.

Table 3.3: Relating interview and workshop campaigns together.

Approach	Activities	+	-
Interviews first	Identify key issues in interviews Discuss issues and agree approaches at workshops	Feels safer Gives attention to different viewpoints	May take time to build up the 'big picture' and discover issues
Workshops first	Explore goals (and scenarios) in workshops; identify issues Analyse specific issues with individual experts	Can quickly come to the heart of a problem	Feels risky, if first workshop is based on a briefing by client but little knowledge of the organisation

Tips for Discovering Goals

- Start with the people you know (your client, for example).
- Find out who the other stakeholders are.
- Use interviews and workshops to find out what their goals are.
- Listen carefully for signs that goals may conflict.
- If people ask for design features, see if you can find an explanation for why they want those features. Asking people 'Why do you want that?' may be too direct for some people. A more indirect approach may work better: 'So that will help you to get [the desired goal]?' or 'What does [that feature] do?'
- 'Play back' to people what you think you have just heard: 'So if I understand you correctly, you want [the desired goals]?' They will often correct you, and add more detail.

Running a Goals Workshop

Activities for a workshop can include:

- setting the scene, introducing the stakeholders who are present, defining the context;
- describing scenarios (see Chapter 5) and identifying goals from them;
- brainstorming goals (in different areas of concern);

- clustering goals, identifying larger/common goals;
- identifying possible conflicts and any terms in need of definition.

These questions can be handled using standard workshop techniques, such as brainstorming, clustering sticky notes and interpreting the clusters thus formed. Media could include the traditional (flipcharts, sticky notes and wall-space) or a more automated approach to capture people's comments and votes.

3.2.6 The Negative Side

Answering the questions:

- What could go wrong (here)?
- What could hostile forces intend here?
- What are you going to do about it?
... so you can handle real problems and work robustly in an untidy world.

The Challenge of the Negative

Much of the effort that goes into the design of systems and products is meant to prevent or mitigate problems of any type, from minor errors to deliberate fraud, from nuisances to catastrophic accidents (Alexander, 2004) [3].

Considering the negative side helps to create more robust requirements; in the case of business-critical and safety-related systems, an unhandled exception means a failure, and possibly an accident.

The negative side also forms part of the rationale (see Chapter 7) for a system or product.

In business-critical and safety-related systems, the negative side is the driving force for security, reliability, and safety requirements and features (see Chapter 6).

For example, a car's steering and braking functions, and railway signalling systems are designed precisely to avoid causing accidents; a bank's security procedures are designed to detect and prevent fraud.

The negative is critical to good requirements. Businesses are generally very well aware of the key risks that they are trying to mitigate. Paradoxically, this means that they may assume these risks are obvious and do not need to be stated explicitly. It can also mean that they tend to overlook new risks created by changing technologies. For example, criminals have been quick to exploit the new opportunities for fraud created by online banking and the online use of credit cards.

What can go Wrong?

Once you understand what stakeholders want your system to do, it is helpful to think about what could go wrong. Analysts call anything that gets in the way of a goal an ‘obstacle’ (Table 3.4).

Table 3.4: Obstacles to your goals.

Kind of goal	Kind of obstacle	Example
Functions in software	Exception event	<p>In software, every programmer’s experience is that a high percentage of the effort of programming comes from error handling: dealing with exception events. An urban legend tells of a system that failed because of software that was delivered still containing the code:</p> <pre>else { /* going to lunch now, must put an exception handler in here */ }</pre> <p>Such stories contain a grain of truth. Dealing with exceptions is difficult; handling all possible exceptions is generally impossible.</p>
Safety	Hazard	<p>In aviation and railways, improvements to safety have historically been driven by responses to accidents. A crash happens, then people investigate the causes, and identify exceptions that nobody had thought about. Can we do better? At least we can say that any exception we find and handle properly is one less way to cause an accident.</p>
Functional	Feature interaction	<p>In telecommunications, adding a new feature may cause other features to break. See the box ‘Feature Interactions’ (page 69) for an example.</p>
Functions using electrical or electronic devices	Electro-magnetic interference	<p>In electrical and electronic systems, any device may harm any other device by radiating power at different frequencies. Many of the considerations described for feature interactions apply – again, it’s an ‘N²’ matrix issue.</p> <p>The problem is so common and so serious that there are standards for emissions (to reduce the risk of harm to neighbouring devices). Look at the base of your keyboard or laptop and you will see various organisations’ logos certifying the device’s compliance. It is also routine to include requirements for devices to have a degree of immunity from radiation coming from other devices.</p>

Table 3.4: (Continued)

Security	Security threat	In any secure system (any government or commercial database or network, actual security systems, military equipment, etc), the risk of people intentionally breaking in and stealing data, money or secrets, or doing damage, is large and increasingly serious.
Reliability, performance	Operational incident	In any large system, reliability and performance suffer at busy times, when components fail, or with accidents or bad weather.
Commercial	Competitive threat	For any commercial product, the actions of competitors and other negative stakeholders (e.g. politicians, the public) may powerfully affect the success of the product in the market.

Identifying Obstacles and Threats

The examples in Table 3.4 show that obstacles come in many forms and affect every product and service. They can cause problems ranging from the trivial to the catastrophic. They consume a large percentage of development effort.

How should you identify obstacles? The good news is that you can start long before code is written or metal is cut. You don't even have to wait for prototypes and simulations. You can start as soon as you know the goals for your system. You can go into more detail as soon as scenarios are defined.

Running an Obstacles and Threats Workshop

A search for obstacles and threats forms an effective workshop topic, once the main (positive) goals have been identified. That leads naturally to a search for ways to mitigate those problems, leading to the discovery of new or missed requirements. These may be for broad safety and security goals, or for more specific responses, such as system functions which have a safety or security purpose.

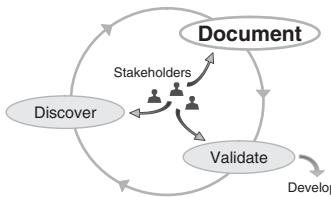
Activities for a workshop can include:

- What threats/obstacles are there to this goal?
- How could environment/weather/other systems threaten our product?
- Which of these goals could conflict with each other?
- What could we do to minimise/avoid this conflict?
- How could we prevent these threats from happening?

- How could we mitigate these threats if they do happen?
- Which of these threats/obstacles/conflicts are the biggest risks for the project?

These questions can be handled in a variety of ways, as for any workshop. For example, you can have a ‘red team’ devise the most effective threats it can come up with, and then have the workshop devise countermeasures to those.

3.3 Documenting Goals



3.3.1 Drawing Goal Diagrams

The interplay of goals that support or conflict with each other can be drawn like the spacecraft goal model (Figure 3.2). The notation is minimal, with simple options for adding structure if you need them.

If possible, start with the top-level goal, which could be the mission of your project. Draw it as a bubble near the top of your diagram. If desired, colour or highlight it as important, e.g. with large text or a thick border, or simply label it ‘Top-level goal’.

Add goals that support the top-level goal beneath it (a lower position tends to reinforce the message ‘these are subsidiary goals’).

Draw an arrow from each supporting goal to each supported goal. If desired, label these arrows with a ‘+’ or colour them blue to indicate support.

Draw an arrow labelled with a ‘-’ to indicate any instances where a goal weakens, obstructs or prevents another goal. If desired, colour these ‘conflict’ arrows red or use a dashed line to show goal conflict.

Add any important obstacles and threats that could interfere with progress towards the goals. Optionally, colour or highlight obstacles and threats.

Add ‘negative’ arrows (drawn as for ‘conflict’ arrows) to show which goals they obstruct and which goals help to overcome the obstacles or mitigate the threats.

Optionally, label goals to characterise their type. This is an easy matter with a goal modelling tool (Figure 3.6) and equally straightforward by hand.

Feature Interactions: Unexpected Obstacles

A simple example of ‘feature interaction’ is what happens when you add ‘call waiting’ to a phone that already has automatic call answering. If a call is in progress and another call arrives, do you route the second call to automatic answering, or do you beep to announce a call waiting? The two features conflict.

Feature interaction means you can’t assume that adding something means just adding some new code. You may have to go through the implementation of many existing features to see what could possibly interact, and hence what needs to be changed.

This leads to an ‘ N^2 ’ matrix: a table of N features that could cause trouble to the same N features (less one) that could suffer. With a large N , the matrix is uncomfortably big. Sometimes problems can be avoided by using simple tricks, like assigning a priority to each feature: a higher priority feature always ignores interruptions from lower priority features, and always gives way to higher priority ones. If that isn’t enough, discovering the requirements for avoiding feature interaction means examining each cell in the matrix in turn.

Feature interaction probably affects any complicated system – it’s just that telecommunications discovered it first (they have large amounts of software, and are very careful).

As shown in Figure 3.6, goals can be labelled with both:

- their requirement class (functional, safety, security, etc);
- their goal/obstacle type (goal, obstacle, threat, key goal).

Figures 3.3 and 3.4 extend the basic goal notation by enclosing the goals that belong to a particular stakeholder role in a circle labelled with the name of the role.

It is probably worth using a tool for this purpose, so that you can:

- select the goals to be shown at different times (i.e., you can prepare different partial views of the goal model);
- rearrange the positions of diagram elements for clarity.

Figure 3.4 incidentally illustrates both of these drawing strategies.

3.3.2 Other Ways of Documenting Goals

You may sometimes need to make it explicit whether a set of subgoals consists of strict alternatives, partial alternatives or necessary contributions

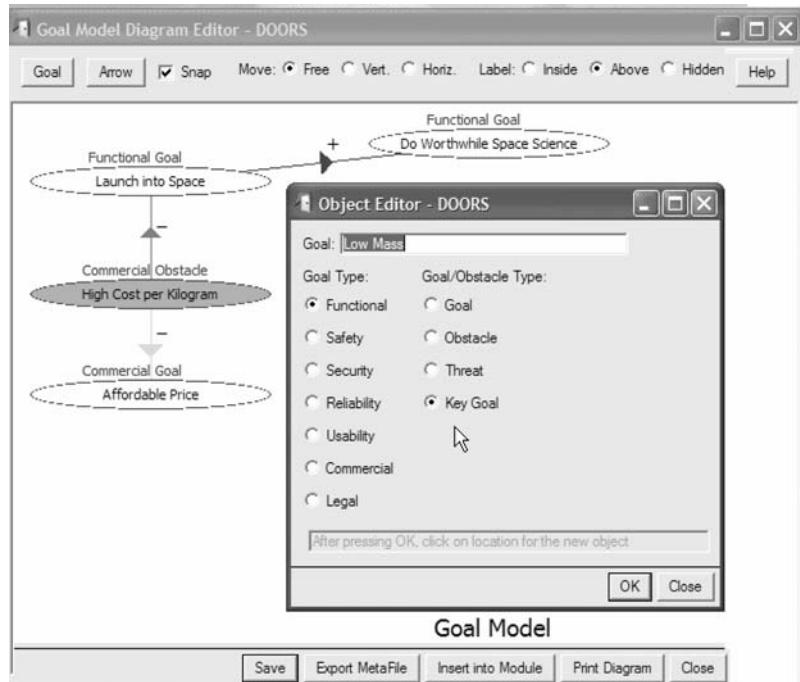


Figure 3.6: Labelling goals with their type.

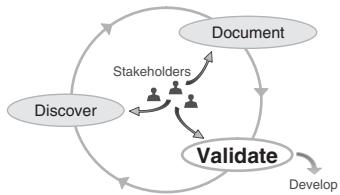
to the parent goal. A simple way of doing this is to draw an arc joining the contribution arrows from the affected subgoals, labelling the arc with ‘or’, ‘and/or’, or ‘and’ respectively. An example is shown in Figure 6.1 in Chapter 6. If you treat ‘and’ (all subgoals are necessary to achieve the goal) as the default case, then many of your goal diagrams will not require the use of arcs.

See Appendix C for further discussion of goal modelling with tools.

Tips for Documenting Goals

- Start with a bulleted list of what people want (see Figure 3.3).
- Draw each goal as a bubble.
- Draw an arrow from each goal to any goal that it helps to achieve.
- Draw an arrow marked ‘–’ from each goal to any goal that it obstructs.
- You may find it helpful to label the goals with their type, (e.g. functional, reliability, commercial, legal, etc).

3.4 Validating Goals



Even more than other requirement components, goals come from people, often directly in stakeholder interviews and workshops. These means, therefore, also offer the best ways of checking that you have identified people's goals correctly, and have resolved any conflicts to everyone's satisfaction. If you still have the feeling after an interview campaign that the goals are not quite right, then a workshop may be the answer.

You can never know if you have a complete set of stakeholders' goals, as it depends on what people want. Incompleteness is guaranteed, however, if you have missed a stakeholder. The best you can do towards ensuring completeness from any one stakeholder is to play their goals back to them, via simple scenarios or prototypes, for example, and to listen carefully for any changes or additions.

Goal models can be treated as complete when:

- they are agreed by stakeholders;
- functional goals are seen to be implementable;
- non-functional goals are either:
 - directly achievable; or
 - analysed into subgoals that are implementable.

Once you have a set of goals, preferably in a goal model, you should check them for any inconsistencies and possible conflicts.

A simple inconsistency, such as having two different values for a target performance, can easily arise in a busy workshop when you scribble goals on flipcharts, or in individual interviews when two or more interviewees state their separate opinions. Such differences can often be resolved informally.

Goal conflicts are not necessarily wrong (unlike conflicts in formal, especially contractual, requirements). You should feel pleased when you discover goal conflicts. They indicate the tensions that will drive design trade-offs throughout the project.

However, goal conflicts can be less benign. They can directly indicate personal or 'political' faultlines within a project. At the very least, these show that the project will be difficult to manage. At worst, they may indicate a need

to restructure the project or cancel it as unworkable. Goals are very powerful, as they are the motivation for everything else.

Ignoring goal problems will not make them go away. They just become more intractable – and much more expensive – as projects move into their detailed design and implementation phases with unresolved issues.

Serious inconsistencies may occur when people favour different strategies; for example, some might favour launching a simple product or service quickly and improving it incrementally, while others favour waiting until the first release can be made really impressive. This sort of conflict could reveal itself indirectly with, for example:

- some stakeholders suggesting numerous novel features for a product;
- others insisting on qualities such as ‘easy to use’, ‘not too fiddly’, ‘reliable’, ‘familiar’.

Your best chance of detecting such a conflict may be in a workshop, where the body language and opinions of stakeholders may make their strategies apparent. Other stakeholders may be aware of such ‘political’ matters but only be willing to talk about them off the record (not in an interview or workshop). Since such unstated things may be the key to understanding a project, you should make use of every opportunity to hear what people have to say informally.

When you have a clear idea of the goals of a project, go over your goal model and check that it accurately expresses the situation. If management action is required to resolve any issues, talk discreetly to the project manager and discuss how the issues might be addressed.

Tips for Validating Goals

- Goals come from people. Check your documented goals with your stakeholders.
- Check that you don’t have any duplicate goals (maybe worded slightly differently).
- If you have found any goal conflicts, check that people know about them and have agreed how to deal with them.
- Watch out for goals that demand absolutes ('100%', 'all users') as these may not be attainable (or even desirable).

3.4.1 Things To Check Goals Against

- Interfaces mentioned in the context model (Chapter 4)
- The purposes of scenarios (Chapter 5)
- Reasons given for decisions (Chapter 7)
- Targets to be measured (Chapter 9)
- Things people ask for in interviews (Chapter 11) and workshops (Chapter 12)

3.5 The Bare Minimum of Goals

Agree what the project is trying to achieve, regardless of the implementation approach. If you can't find this out, something is seriously wrong.

3.6 Next Steps

- Analyse stakeholder influences (Chapter 2).
- Create scenarios (Chapter 5) for functional goals.
- Document technical terms in a project dictionary (Chapter 8).
- Analyse the required qualities and constraints from non-functional goals (Chapter 6).

3.7 Exercises

1. You are a product manager for a machine tool company. The directors have asked you to develop a new cutting machine to cut cloth for fashionable dresses of all sizes and patterns. The machine will be sold to clothing makers around the world.
 - a. What are the major goals for this project?
 - b. Using the list of stakeholders for this project (see the exercise in Chapter 2), identify the likely sources of tension (possible conflict) between stakeholders' goals.

2. Model the conflicts between stakeholders' goals for the restaurant examined earlier in this chapter. Start by extending Figure 3.4 to cover the other goals shown on Figure 3.3. Either use a tool or a large sheet of paper.

3.8 Further Reading

3.8.1 Goals

1. Sutcliffe, A. (2002) *User-Centred Requirements Engineering, Theory and Practice*, London: Springer.
We do not know of any good industrial books that major on goal modelling as a way of finding requirements, but this is a good academic book that is helpful on goal and task analysis.
2. Lauesen, S. (2004) *User Interface Design, A Software Engineering Perspective*, Harlow: Addison-Wesley.
User interface design is a profession not far from requirements work. Lauesen's excellent book shows in a practical way how to use goals and tasks to create a user interface that works well for people.

3.8.2 The Negative Side

3. Alexander, I. (2004) Negative Scenarios and Misuse Cases. In Alexander I. and Maiden N. *Scenarios, Stories, Use Cases*, Chichester: John Wiley & Sons, Ltd
There is an extensive literature on security but few books, if any, on how to discover requirements for it. This book includes an entire chapter on the negative side.

3.8.3 The i* Goal Modelling Notation

The i* literature is mainly academic; there is little straightforward industrial guidance on the approach. The ungoogleable name has perhaps not helped, though a search for 'istar goal modelling' does now work.

The home page of i* on the web, listing many research papers, is:

4. <http://www.cs.toronto.edu/km/istar/>

A set of tutorial slides on i* is available, linked from [4], at:

5. Part 1: <http://www.cs.utoronto.ca/pub/eric/tut1.1-v2.ppt>
- Part 2: <http://www.cs.utoronto.ca/pub/eric/tut1.2-v2.ppt>
- Part 3: <http://www.cs.utoronto.ca/pub/eric/tut2-ER01.ppt>

CHAPTER FOUR

Context, Interfaces, Scope

Any system or piece of work responds to things that happen outside it.

Suzanne and James Robertson

Requirement Elements	Context, Interfaces, Scope						
	Priorities	Measurements	Definitions	Rationale and Assumptions	Qualities and Constraints	Scenarios	
Discovery Contexts							
Introduction							
From Individuals							
From Groups							
From Things							
Trade-Offs							
Putting it all Together							

Answering the questions:

- What should be included, and what left out?
- Where is the boundary of this system?
- What are the boundaries of this product?
- What approach do you need to define your scope?
 - ... so you know what you have to cover, and what you don't
 - ... so you adopt an approach capable of reaching a solution.

4.1 Summary

The scope of a project is what it is to include: everything that is inside the boundary of its product. But projects have to discover for themselves where their boundaries are, so the task is dangerously ill-defined. 'Soft systems' work deals with such difficult issues, including political, social and economic forces, until enough agreement is reached for 'hard' traditional systems engineering to take over safely.

One of the biggest risks for a project is for its scope to 'creep' (to keep changing), with nobody then sure what is included, how long the work will take or what it will cost. The result is that it delivers the wrong product, late and over budget, if it completes at all. Probably more projects run into trouble for this reason than for any other.

The solution is to take time up front, early in the project, to define the scope of each product. That means understanding the product's context and interfaces in enough detail to be able to make scoping decisions accurately and lastingly.

This chapter is divided into two halves:

- In the first half, it looks at how to translate an ill-defined problem with no definite boundary into one with a known scope that a project can realistically take on. This is mainly a 'soft', human-facing activity.
- In the second half, it looks at how to proceed systematically from the newly-defined boundary, to identify in turn the interfaces, events and event-handling requirements. This is mainly a 'hard', analytic activity.

4.2 Introduction

Projects do not begin with a neat textbook challenge: 'Fill this template with 47 pages of requirements of the following kinds, and you're done'. You don't even know what to specify. The problem itself is undefined. If it were a Sudoku puzzle, you would have to begin by working out how many squares there

were to fill in. Conventional problem-solving techniques and algorithms can't be applied to this sort of situation. In many cases, a much 'softer' approach is needed, before 'hard' and definite engineering techniques can be brought to bear.

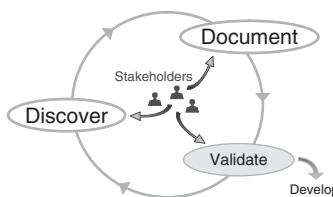
In this chapter, therefore, we look at ways of bringing the boundaries of systems and products into sharp focus. On your project, the boundaries may already be quite well defined, in which case you can pay attention to the 'harder' techniques.

This chapter looks in turn at:

- a 'soft systems' approach for ill-defined boundaries (section 4.3);
- switching to a 'hard systems' approach for known events (section 4.4).

These are treated as two separate discover-document-validate cycles.

4.3 A 'Soft Systems' Approach for Ill-Defined Boundaries



The idea of a 'soft system' was introduced by Peter Checkland in his *Soft Systems Methodology* (Checkland and Scholes 1999) [1]. A soft system is one that involves social, political and emotional issues as well as technology: again, not just products or services but people, procedures and all the relationships between people that make real life complicated but practical. To quote Checkland:

'Hard systems engineers tackle rather well-defined problems, while soft systems methodologists address messy, ill-structured, problem situations.'

But Checkland goes much further than this. His soft systems methodology (SSM) encourages you to change the way you look at the world, whereas 'hard' systems engineering gives you a fixed way of looking at a problem. SSM invites you to wonder if you are wrong, and gets you to explore different models and possibilities.

For development projects, one of the most helpful ideas suggested by Checkland is the 'rich picture', a diagram of what is happening in a business.

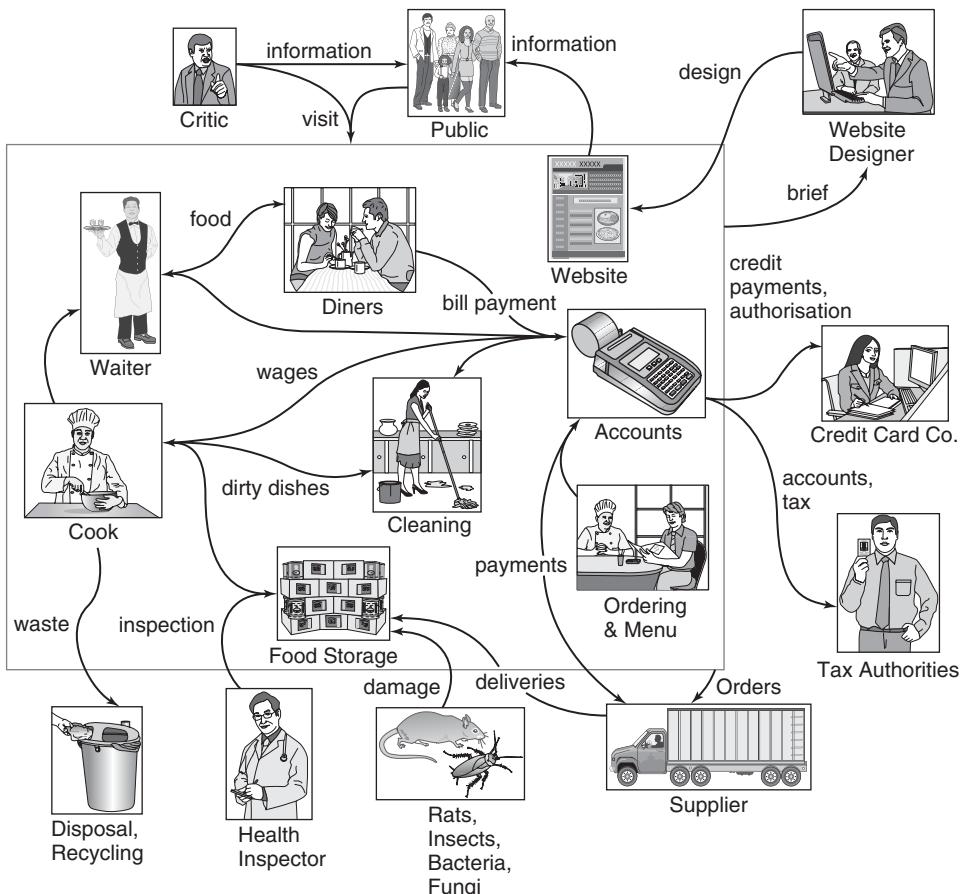


Figure 4.1: Rich picture of the work of a restaurant.

You can, for example, draw an informal but very useful view of a system's context and scope (Figure 4.1), including any concerns and issues mentioned by stakeholders.

4.3.1 You are Part of the Soft System you are Observing

A rich picture can describe anything you want to show about how a business is, or how it should change. This can include your intervention and where that intervention should be made. It's a bit like modern physics: you aren't a classical scientist objectively observing a distant phenomenon; as an engineer or consultant, you are part of the soft system that you are trying to improve. In quantum mechanics, the act of making an observation itself changes the state of the observed particle from undetermined (e.g. its spin is both up and down)

to a definite state. In a soft system, any intervention, such as introducing new software or ‘streamlining’ a business process, inevitably becomes part of the whole way the business works.

‘Hard’ versus ‘Soft’

Soft systems people can be quite critical of ‘hard’ systems engineering, in which they include many software engineering approaches. Note that software with its definite interfaces, business rules and algorithms is ‘hard’ in SSM terms, falling into the category of systems, not the messy world.

The enormous success of engineering and information technology (IT) has greatly expanded the use of ‘hard’ systems, but it has also caused a wide range of ‘soft’ problems. For example, governments introduce ‘hard’ IT systems to solve specific problems, such as supporting people who are sick or unable to work. These often fail for ‘soft’ reasons, such as people finding the systems too complicated to use, or criminals finding the systems very convenient for fraud.

The good news is that soft systems work is an excellent way of leading into a development project. Businesses and governments are starting to realise how important it is to begin by understanding the context.

Soft systems work can be much wider than identifying the need for a product. In fact, the more widely you draw the boundaries of a system (as in Figure 4.4), the ‘softer’ the system becomes. An organisation should really do some serious soft systems thinking to work out what ‘hard’ systems and products it needs, before it dives in to develop them.

4.3.2 From Stakeholders to Boundaries

Chapter 2 showed that people who will never operate the product under design (who ‘never touch a keyboard’) are, nevertheless, extremely important as stakeholders in development. For example, regulators and company directors exert different pressures on a product manager. In other words, people in the outer layers of the ‘onion’ have a powerful impact on the design of a system. Let us look at what the onion model can show us about a system’s boundaries.

In popular usage, ‘system’ means almost any kind of product, process or procedure. We will try to use ‘system’ more carefully to mean a thing (under development) able to work autonomously to deliver a set of functions.

Tips for Rich Pictures

There are no rules for drawing rich pictures, other than to show whatever needs to be shown. Figure 4.1 is actually quite formal as rich pictures go, because it was drawn by a requirements person. Feel free to show:

- influences from stakeholders who are not involved directly in operations;
- issues and concerns, such as things that are unclear;
- the process you and the stakeholders agree to follow;
- pressures and forces that may slow the work down;
- things that don't seem quite nice, like rats and rubbish. A business like a restaurant definitely can't succeed just by getting its hygiene right; but it can certainly fail if the public health inspector closes it down for getting its hygiene precautions wrong.

Read about the soft systems methodology, and reflect on what it means for your work.

Few *products* are really autonomous, as they are mostly operated by people, who follow rules and procedures to use the products correctly (often, to deliver a service). Even products that seem to work all by themselves, like some spacecraft and factory robots, are typically installed, configured, tested and maintained by human operators.

So, a system usually consists of several things working together:

- one or more products; and
- some people (operators of different types); and
- rules or procedures that say what to do in different situations.

Table 4.1 gives some examples of systems, their human operators and some of the products the systems are composed of. There are often many operators and products in one system, and these often combine to provide a service.

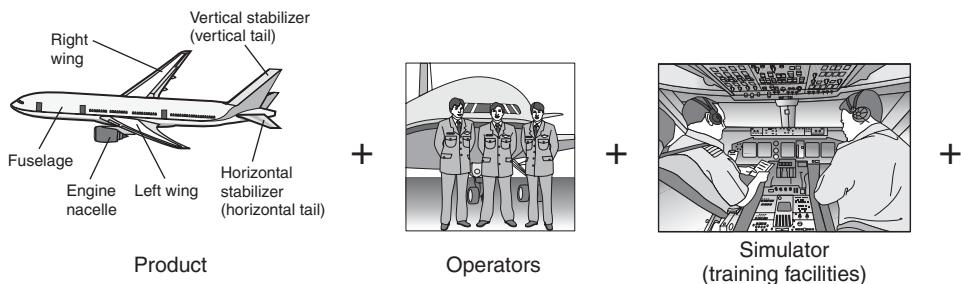
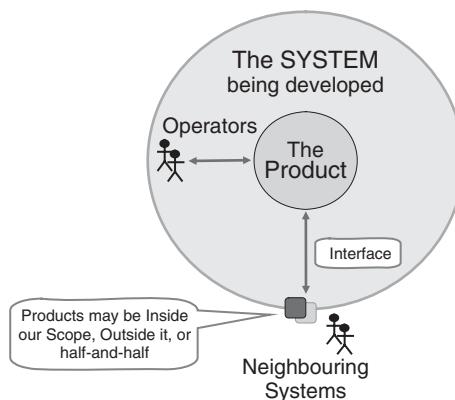
Systems may include more people than their product's normal operators, such as a plane's crew. Other operators may be involved occasionally, for routine or emergency maintenance, for instance. Also, operators have to be trained and perhaps tested. Training facilities, such as flight simulators, may also form parts of a system (Figure 4.2).

As a general rule, therefore, the system boundary is outside the product boundary (Figure 4.3). The system contains more than just the product.

It is often quite hard to identify exactly what the boundary of a system should properly be. Is an aircraft with its aircrew (Figure 4.4) the right choice,

Table 4.1: Systems, operators, products and services.

System	Operators within system	Products within system	Service provided by system
Restaurant	Cook, waiter, cleaner, etc	Cooker, fridge, etc	Convenient and elegant dining
Manned aircraft	Pilot, cabin crew, etc	Aircraft	International transportation
Railway line	Driver, signaller, ticket seller, etc	Train, track, stations, signals, etc	Transportation

**Figure 4.2:** System = Product + Operators + ...**Figure 4.3:** System is bigger than product.

or should the boundary be drawn somewhere else? Are the maintenance facilities and crew inside, outside or actually on the boundary, for example?

Depending on the situation, the project might choose to widen its scope to include more capabilities within the boundary.

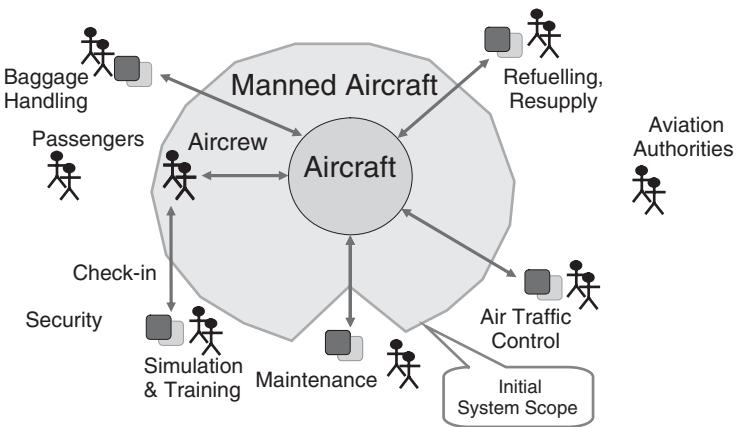


Figure 4.4: First attempt: System = Manned Aircraft?

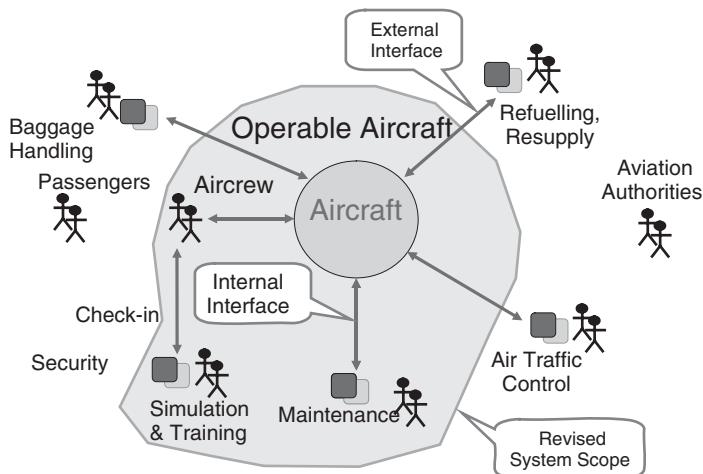


Figure 4.5: Second attempt (wider scope): System = Operable Aircraft.

For example, a new startup airline could take advice from the aircraft maker on all the facilities it would need in order to get flying. The aircraft maker might sign a contract to deliver not only aircraft to the airline but also spares, maintenance equipment and simulators. The aircraft maker might go so far as to build, equip and perhaps operate the maintenance and training facilities to its own standards, in return for a much bigger profit than could be made on the aircraft alone ('Operable Aircraft' in Figure 4.5).

Different stakeholders may have overlapping or contrasting views of system scope. For example, the point of view of an airport and that of air traffic control both include aircraft. The airport is also interested in baggage handling and

refuelling/resupply, while air traffic control is interested in radar and weather forecasts. There is no ‘right answer’ that will be appropriate for everybody.

4.3.3 Identifying Interfaces

Ignoring interface issues makes them bigger.

Ivy Hooks and Kristin Farry [4]

By bringing together the points of view of the relevant stakeholders, such as aircrew and maintenance, you will come to important decisions and trade-offs about where to draw your system’s boundary. Those decisions will determine the system’s scope and interfaces.

Continuing with the example of Figure 4.5, let us consider how the system’s interfaces are affected by extending its boundary.

Because maintenance is inside the ‘operable aircraft’ system boundary, the interface between maintenance and the aircraft is now internal. It will make

Interfaces aren’t just Hardware

Probably the most common everyday use of the word ‘interface’ is as a connector to a piece of computer hardware. People talk about a Firewire interface, a USB interface and so on.

The interface between a printer and a computer is not just a cable, however. The data and message formats, and how these are to be handled, are equally vital. For example, printer and computer must agree about when the printer is supposed to wait for a command, and when it should act on the information it has received. **Behaviour** therefore often forms part of the interface definition.

Interfaces **need not be for data and wires**. For example, your car interfaces to the pump on a filling station forecourt via a simple circular hole. With the rise of diesel engines (now up to 44% of new car sales) and the lack of standard colours (diesel nozzles used to be coloured black, but now they are often green), there is an increasing problem of misfuelling. Lead-free petrol (gasoline) nozzles are smaller than diesel nozzles, making it hard to put diesel into a petrol engine, but the interface fails to prevent petrol from being put into a diesel engine – with disastrous consequences. One proposal is for an oval diesel nozzle, so the two fuels would have safely incompatible interfaces.

Interfaces **need not be between machines**. Human-machine interfaces have a human on one side, while many business interfaces still have humans on both sides.

sense to think of both maintenance and simulation/training as subsystems of the 'operable aircraft' system.

The project will now have complete control over the design of the training simulator and of the maintenance interface. The project will be free to choose, for example, whether the maintenance interface should be:

- the same as for other aircraft (i.e. commonality), to minimise the need for new tools, equipment and training or
- specially designed to suit the new aircraft, to maximise efficiency of maintenance (and hence to increase the availability of fully-maintained, ready-to-fly aircraft).

This is an example of a trade-off (see Chapter 14), in this case between a design for commonality and a design for maintainability and availability. These are key qualities (see Chapter 6) of a product or system. Desired qualities often have effects throughout a system, so decisions about such qualities are far-reaching.

Interfaces, whether internal or external, can be of different kinds, and will be treated differently during system development (Table 4.2).

It is not sufficient only to define product interfaces, as you would then overlook interfaces between people. These may not be automated, but they can be critical to a system's operation. Of course, this is another way of saying that you need to take a wider and 'softer' look at your whole system, rather than concentrating only on your product.

4.3.4 Documenting Interfaces

Interfaces are the places where your product meets the outside world or, more specifically, connects with other products (hardware or software).

When your product is part of a larger system, the interface requirements are the mechanism by which the system design is transmitted 'down' into the design of your product. In particular, internal interfaces between your product and others created in the system design become external interfaces from the point of view of your product.

Interface requirements involve both functionality (product behaviour) and design, and may need to be specified in full design detail. Luckily, however, many interfaces are standardised. Interfaces can therefore usually be specified exactly, just by calling out the relevant section of the appropriate industry, national, or international standard.

In the following examples below, the GPS (Global Positioning System) specification is an American project document that functions as a de facto international standard; USB (Universal Serial Bus) is an industry standard under the control of the USB Implementers Forum, whose members include Apple, Hewlett-Packard, Microsoft and Intel.

Table 4.2: Handling different types of interface.

Type of interface	Example	How to discover this	Treatment
Between people	Aircrew report a problem to maintenance staff	Study the 'soft' system and its business processes (this chapter) Scenarios (Chapter 5)	Write a procedure; train crew to follow the procedure (i.e., write a training course, and run it)
Between people and products (These can be divided into inputs and outputs)	Aircrew interact with controls on the aircraft, both giving commands and receiving information	Scenarios (Chapter 5) Prototyping (Chapter 13) Similar/existing/rival products (Chapter 13)	Specify the user interface, considering effect on work (ergonomics); write a procedure for each use of the interface; train crew to follow the procedure (i.e., write a training course, and run it)
Between products (These can be in either direction, or both)	Automatic equipment diagnoses a fault on the aircraft	For interfaces to existing external systems: standards, or manufacturers' data sheets For interfaces to existing products/systems: speak to your developers/suppliers For new interfaces: iterate requirements and design (Chapter 14)	Specify the interface (the data and other quantities to be exchanged, and the required behaviour), using standards if possible; impose the interface as requirements on both products

Interface requirements may need to cover:

- how to handle incoming information or physical quantities (electrical, gases and liquids, etc);
- how to provide outgoing information or physical quantities;
- physical connectors.

Many interfaces are bidirectional, i.e. both incoming and outgoing.

Here are some examples.

Incoming: *The system shall receive GPS signals as specified in the (NAVSTAR) GPS SPS Signal Specification, 2nd Edition, 2 June 1995.*

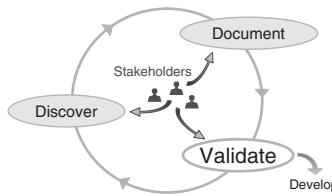
Comment: This is to enable positioning and navigation.

Outgoing: *The system shall provide a 5.0 Volt, 350 milliAmp power supply in each USB-2 connector as specified in the PoweredUSB Battery Charging Specification 1.0.*

Comment: This is intended to be suitable for recharging a mobile phone.

Physical connectors: *Four USB-2 sockets shall be provided on the front of the console.*

4.3.5 Validating your Choice of Boundary



Drawing the boundary of a system or product seems simple, but it is undoubtedly among the most difficult and critical tasks for a project to discover and validate.

- It is **difficult, because:**
 - it requires skill and creativity to explore alternative possibilities for a boundary; no prescriptive method exists to say ‘this is how you choose the best boundary for your system’;
 - the problem is ill-defined; you don’t know what a project has to do until you have a boundary, but you don’t know where to draw the boundary until you know what the project is supposed to be doing. This is the nature of soft systems work.
- It is **critical, because** all later work on the project is affected by the choice: what products to develop, what interfaces, what events to handle, and hence what the design is and what risk is involved.

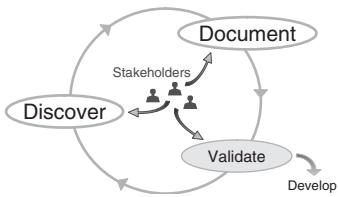
If your system and product boundaries are uncertain then, clearly, more effort is needed to define them.

If, on the other hand, a definite boundary is drawn but most of the system’s work involves repeatedly crossing the boundary to achieve anything, then the boundary probably needs to be rethought.

Completeness is probably not fully definable for a boundary – in many cases, it is a matter of business choice whether something should be included or not.

A boundary is certainly incomplete if an important stakeholder has been forgotten, or if key goals have been overlooked. Where the goals include providing interfaces to other systems, that can be checked systematically.

4.4 Switching to a ‘Hard Systems’ Approach for Known Events



In this section, we change gear from the mainly human-facing ‘soft systems’ approach to a more traditional systems or software engineering approach. This shows how to define functional requirements to handle the events identified from the agreed context and interfaces.

4.4.1 The Traditional Context Diagram

Once we have a good idea of the work to be done, we can draw a traditional context diagram (Figure 4.6) to define the scope of the work as a whole. The scope is drawn as a circular boundary, showing graphically what is outside the project’s remit. This directly helps to bring issues into the open, so it tends to prevent scope ‘creep’ (where undiscovered issues emerge slowly and inconveniently without prompting).

The context diagram intentionally ignores anything that happens outside the boundary that does not directly involve an interface (a labelled arrow) with the system or product in question.

This represents the move from an ill defined problem needing a lot of ‘soft’ work, to a relatively well defined problem where straightforward analysis can be applied. It is dangerous to do this before the boundary has been properly explored.

It is well worth listing the things that you have agreed are out of scope, as this saves you from having to keep discussing them in every project meeting.

4.4.2 Scope as a List of Events

An arrow into the circle on a context diagram means an ‘event’, the typically unpredictable arrival of a signal, to which the system has to respond, by carrying out one or more tasks. The in/out list shown in Figure 4.6 is then a list of events our system has to handle or, equivalently, the tasks that we agree our

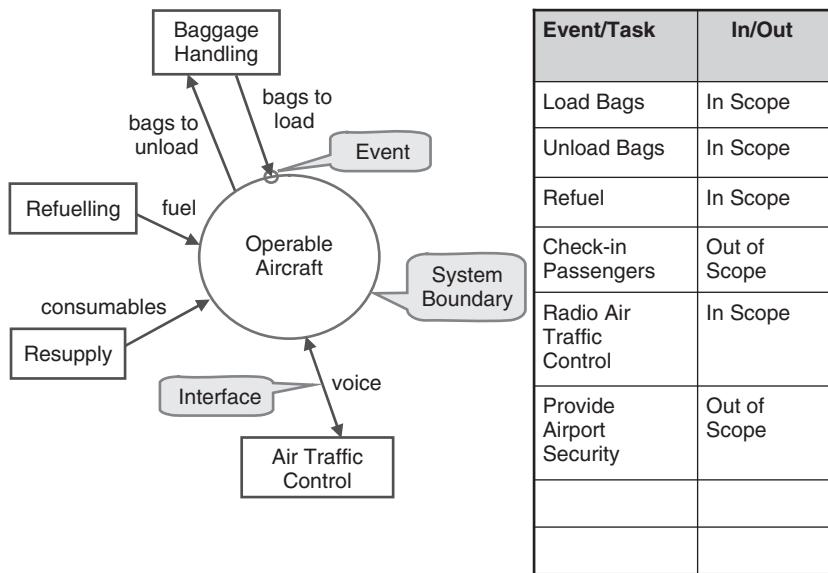


Figure 4.6: Traditional context diagram and in/out list under development.

system will have to carry out (to handle the events). With this understanding, we can now formalise 'scope' as the set of events we have decided to handle.

Handling each such event becomes a requirement if the event is agreed to be in scope.

Events that consist of something coming into the product are of two types:

1. External (data or physical) event: the unpredictable arrival of something that the product should use as a stimulus, i.e. it should wake up and do something in response. The stimulus can be:
 - a message (like a packet of data);
 - a signal from a sensor (e.g. the pressure in a pipe has fallen below a threshold value, or the detected movement or body heat of an intruder); or
 - an explicit control input (e.g. a button press, a mouse click, a touch on a touch-sensitive screen). Control inputs are handled separately in some modelling approaches.
2. Time-triggered event: a time signal, e.g. from a shared clock on a network or data bus. This acts in much the same way, triggering the product to do something in response, as if a stimulus had arrived from outside.

Either way, being able to handle each incoming event that is agreed to be in scope is the subject of a functional (event-handling) requirement.

Notice that a *business event*, like the arrival of bags to load, is often translated into a *product ('system') event*, like the pushing of a button to start a conveyor belt.

4.4.3 Expressing Event-handling Functions

Two alternative styles can conveniently be used to express event-handling functions:

- textual event-handling requirements;
- condition-action tables.

Let us look at each of these in turn.

Event-handling, or 'When' Requirements

The format for a traditional event-handling requirement is:

When <event is detected>, do <required action>

Similarly, a suitable format for time-triggered events is:

Every <time period>, do <required action>

For example:

When pressure in Main_Input_Pipe falls below Minimum_Input_Pressure, run Main_Pump at Full_Power.

Every Sample_Period, send Cylinder_Temperature to Engine_Controller.

When Payment_Confirmation is received, set Order_Status to To_Be_Despatched.

A similar style can be used for event-triggered business rules:

If Account_Balance exceeds Private_Banking_Threshold for Qualifying_Period then issue Private_Banking_Invitation to Customer.

It is generally a bad idea to sprinkle actual values in requirements ('Every 100 milliseconds, ... ') as these may change. Symbolic names make dependencies clear – if the Sample_Period is 100 milliseconds, then the reader knows that each requirement naming that period is to be handled at the same rate. If you need different sampling rates for different parts of your system, then you simply create two or more symbolic variables: Engine_Sample_Period, Non-Critical_Sample_Period, and so on.

Notice that these styles are appropriate when the architecture of the solution is already known (post-optioneering: see Chapter 14). This can be from the start

of a project, e.g. when you are specifying the software control subsystem of a larger product. In that context, you can refer directly to named components (capitalised in the example). The names should be defined in the project dictionary (see Chapter 8).

Condition-Action Tables

A suitable format for a table-driven requirement is shown in Table 4.3. Tables are often easier to read and to validate than traditional textual requirements. This approach permits formal (mathematical) checking for completeness.

There can be any number of ‘condition’ columns; an empty cell means no (extra) condition, or ‘true’ to a logician. Conditions in a row are logically ANDed together, i.e. you only do the action at the end of a row when *all* the conditions in that row are met.

Table 4.3: A condition-action table.

When condition₁	and condition₂	then
<i>Pressure (Main_Input_Pipe) < Minimum_Input_Pressure</i>		<i>Run Main_Pump at Full_Power</i>
<i>Pressure (Main_Input_Pipe) >= Minimum_Input_Pressure</i>		<i>Main_Pump Inactive</i>

Similarly, there can be any number of rows; each row represents a condition-action rule, i.e. an event-handling functional requirement. Rows are logically ORed together, i.e. you obey *every* rule whose conditions are met. That means that you have to ensure that the conditions of rules that you intend to be exclusive (e.g. you either run the main pump or you don’t) must have mutually exclusive conditions.

Table-Driven Requirements, Table-Driven Software

Tables may be used for many kinds of requirements in almost any kind of system (not just control systems, as Table 4.3 might seem to imply). For instance, authorisation rules can be tabulated (Table 4.4) and handled more or less as a condition-action table.

Table 4.4: Presenting authorisation rules in a table.

Expenditure level (£)	Authorisation required
0.00 – 99.99	Declare on timesheet
100.00 – 999.99	Line manager
1000.00 – 9999.99	Senior manager
10,000 +	Director

Table 4.5: Presenting event-triggered business rules in a table.

Trigger conditions ('if')	Actions ('then')
<i>Account_Balance exceeds Private_Banking_Threshold for Qualifying_Period</i>	<i>Issue Private_Banking_Invitation to Customer</i>

“The ‘If . . . then . . .’ style of business rules” can be handled with equal ease, using a table with the trigger conditions on the left and the actions on the right (Table 4.5).

A further advantage of presenting requirements as tables is that software itself can be table-driven; an interpreter just has to read and obey the requirement tables. This enables values to be updated in separate files or a database without changing or recompiling the software itself. However, changes to such values are as critical as changes to software code; they can cause systems to fail. The boundaries between code, data, requirements and day-to-day business procedures can become very thin and debatable.

Calling up Scenarios instead of Single Actions

Since you often need to take several actions together to achieve a result, it is often convenient to write a scenario (see Chapter 5) to define a sequence of actions. The name of the scenario can then be called up as the ‘action’ part of a condition-action table or when-requirement.

Output Events

So far, we have discussed input events, received from the world outside.

Happily, much the same treatment can be given to output events, those that the system under design generates to affect the world outside.

Events that a product has to generate can be of any kind – printing a document, recording a log entry in a database, displaying a message on screen, sounding an alarm, sending a message, and so on.

The traditional style for a function or capability to be provided to a human operator of a product is:

The <operator role> shall be enabled to <do the required action>.

For example:

The retail sales assistant shall be enabled to print a sales receipt.

The equivalent scenario step in a use case is simply:

The retail sales assistant prints a sales receipt.

The traditional style for a functional requirement for a product is:

The <product> shall <do the required action> <with abc performance>.

For example:

The point-of-sale equipment shall print a sales receipt within 10 seconds of receiving a print request.

See Chapter 9 for measuring the performance of functions.

Context and Use Case Diagrams

1. Context diagrams show interfaces, not functions: People familiar with use cases can find the context diagram difficult. It introduces the idea of an interface, which implies some degree of design, whereas a use case diagram just shows actors (stickmen icons) and functional goals (bubbles).

It may help to point out that use case actors can be ‘intelligent’ systems (those with the power to make simple decisions), so the use of a humanlike icon is a bit misleading. In fact, use case diagrams and context diagrams therefore agree that the product under design can interact with both people and neighbouring systems. However, use case diagrams don’t say anything about the interfaces for those interactions.

2. A use case diagram is not very helpful as a context diagram: A use case diagram lists in outline all the things that a product will have to do. People familiar with use cases may feel that a use case diagram can therefore serve as a context diagram. However, when systems contain people and perhaps several products (not only software), a use case diagram can’t say much about the context and interfaces involved. For instance, people generally do not draw purely human-to-human interfaces and functions like ‘ask for technical assistance’ as use cases. Use case diagrams do not provide a way to indicate known interfaces with other systems, either. Use Case diagrams tell only part of the story.

4.4.4 Strengths and Weaknesses of Context Diagrams

We can now state both the strengths and the weaknesses of the traditional context diagram:

- **Its strength** is that it forms a very convenient summary of the interfaces that a system or product requires, and hence its scope, once the boundary is determined.

- Its weakness is that it is not a very good tool for identifying just where that boundary should be drawn. For that, you need to examine a much wider variety of stakeholders and possibly-neighbouring systems. You can then start to reason which of those should be included, wholly or in part, within the boundary.

This means that, unless the context of your project is already very sharply defined, you should start by looking at the wider, ‘softer’ context of your system, its stakeholders and neighbours. You should then consider where to draw your system’s boundary. Only then should you try to fix an exact list of interfaces.

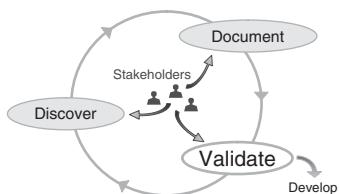
When are you Ready to Draw a Context Diagram?

- When you understand the ‘soft’ system well enough to know you have not missed anything important.
- When the purpose and boundaries of your system are well defined.
- When, therefore, it is safe to move to a ‘harder’ description.

For instance, you are making the software to control a car’s entertainment system. Its external interfaces have already been defined by the car maker. Your task is not to discover new external interfaces – unless one has clearly been missed – but to analyse the required behaviour to create a system (a set of components with ‘internal’ interfaces between them) that works as specified.

You may never be perfectly ready. ‘Soft’ issues and new requirements never stop appearing: stakeholders change their minds; technology advances; politics intrudes. So you can’t wait until all issues are resolved.

4.4.5 Validating Interfaces and Events



- Check that the list of events covers all the interfaces on your context diagram.

- For standard interfaces, ensure that the exact version of the relevant standard is identified.
- For custom interfaces to other systems, agree all details of each interface with the owners of the other system.
- Each event should be handled by when-requirements or their equivalent in condition-action tables.
- The list of alternative conditions in condition-action tables or when-requirements should cover all the possibilities exhaustively.
- Check that each scenario referenced in an event-handling requirement is defined, e.g. as a use case.
- Check that each interface is defined in the data dictionary.

Discovering Missing Functions

You might discover a missing requirement when, on the basis of clues from your interview campaign, you create your own interpretation of your findings. When you show this at the next interview or workshop, someone might mutter that they think there's more to it than that, and refer you to somebody in the commercial department, as illustrated in Figure 4.7.

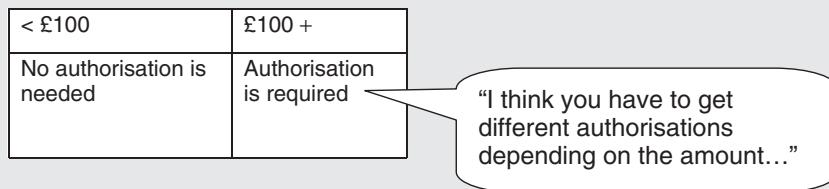


Figure 4.7: Discovering that your model is incomplete.

That is a warning signpost that your understanding is incomplete. Your task is to get to the bottom of the mystery, by finding out who really knows about the situation. It's also possible that asking the question will result in the issue of a new table of authorisations by the commercial department, which may be no bad thing.

4.4.6 Things To Check Context and Interfaces Against

- Stakeholders who could be neighbours (Chapter 2)
- Goals (Chapter 3)
- Scenarios (Chapter 5)
- Things mentioned in interviews (Chapter 11) and workshops (Chapter 12)

4.5 The Bare Minimum of Context

- Make sure you understand your context well enough to agree what your project will cover, and what you will safely leave to your neighbours (knowing they will cover it).
- Make sure you know what events you have to handle.
- Make sure you know that the scope defined by your boundary is realistically achievable.
- Document what is agreed to be out of scope.

4.6 Next Steps

- Create scenarios for the events you have identified (Chapter 5).
- Start collecting and defining terms that you notice (Chapter 8).
- Plan how you will gather your requirements, whether from individuals (Chapter 11), groups (Chapter 12) or things (Chapter 13).
- Plan when to define the interfaces you have identified in full detail; usually, the earlier the better.

4.7 Exercise

Choose a style of restaurant and business model (for example, an elegant setting with independent chef; fast-food pizzas and cola; good coffee and cakes with free Internet access, etc):

- a. Develop a context model for your particular type of restaurant, starting from the (generalised) rich picture of Figure 4.1. Define carefully what you need to control and include, and what you will obtain from other businesses.
- b. List the main events your restaurant's IT system will need to handle.

4.8 Further Reading

4.8.1 Soft Approaches

1. Checkland, P. and Scholes, J. (1999) *Soft Systems Methodology in Action*, Chichester: John Wiley & Sons, Ltd.
2. Beyer, H. and Holtzblatt, K. (1998) *Contextual Design, Defining Customer-Centered Systems*, London: Morgan Kaufmann.
See the Further Reading section of Chapter 1 for comments.

4.8.2 Event-Driven Approaches

3. Wiley, B. (1999) *Essential System Requirements: A Practical Guide to Event-Driven Methods*, Reading, Mass: Addison-Wesley.
Bill Wiley's book is becoming dated but remains interesting as one of the few books to take an event-driven approach to requirements discovery, with many practical suggestions and examples.
4. Hooks, I.F. and Farry, K.A. (2001) *Customer-Centered Products: Creating Successful Products Through Smart Requirements Management*, New York: Amacom.
Hooks and Farry have written a smart and funny book for business people, making the case for doing requirements properly – by which they mean finding out what customers want, and doing that.
5. Lavi, J.Z. and Kudish, J. (2005) *Systems Modeling & Requirements Specification Using ECSAM: An Analysis Method for Embedded and Computer-Based Systems*, New York: Dorset House.
Lavi and Kudish have created a carefully crafted, solid textbook on event-driven systems analysis. You could argue that they start from roughly the point where the current book leaves off, with a sharply-defined business need.

4.8.3 Writing Requirements

6. Alexander, I. and Stevens, R. (2002) *Writing Better Requirements*, London: Addison-Wesley.
See the further reading section of Chapter 1 for comments on Alexander and Stevens' book.

CHAPTER FIVE

Scenarios

The scenario-building exercise utilised the fact that the human imagination is the cheapest multimedia prototyping system around.

Just a few words serve to call up vivid pictures and imagined interfaces.

Jakob Nielsen

Requirement Elements	Scenarios	Priorities	Measurements	Definitions	Rationale and Assumptions	Qualities and Constraints
Discovery Contexts						
Introduction						
From Individuals						
From Groups						
From Things						
Trade-Offs						
Putting it all Together						

Answering the questions:

- How will people use this product or service?
- How do you do this?
- How do you imagine this will work?
- What steps are involved?

... so the product or service does what is wanted, works, and is usable.

5.1 Summary

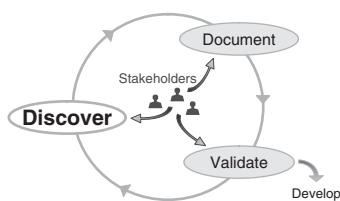
A scenario is a story adapted and structured for engineering use. The purpose of a scenario is to communicate a situation, usually as it evolves through time, in a series of steps.

Scenarios communicate requirements very effectively – sometimes almost on their own – as storytelling is such a natural and powerful way of explaining needs in human terms. Increasingly, software functions are specified entirely as the scenario structures known as use cases.

When scenarios describe interactions of humans and products across ‘user’ and ‘system’ interfaces, they necessarily assume a degree of design: that the system’s work will be divided up at the described interfaces.

Scenarios can be at any level of design detail, from simple descriptions of things that people want to do, to the interactions of components (such as software services) collaborating to achieve a goal.

5.2 Discovering Scenarios



The basic way to find out what people do in their present or future work is to ask them to describe it as if they were doing it – or as they would like to do it.

Discovering What Scenarios to Write

At the start, even determining which scenarios to write can be tricky. Initial leads can come from:

- stakeholders' comments in interviews (Chapter 11);
- goals (Chapter 3) in workshops (Chapter 12);
- known scenarios, e.g. problems with existing systems (Chapter 13);
- study of the system context and scope (Chapter 4);
- standard scenario patterns (e.g. DILO and CRUD, in this chapter).

Generally, the right level of scenario to start with is a **complete, end-to-end story** that deals with how a human operator uses a product to get results. For instance, in an online bookstore, 'buy a book' is a complete story, whereas 'add item to cart' is a minor detail.

It is also generally best to start with **normal, 'happy day'** scenarios, and leave the trickier work on problematic cases for later.

But often, the best guide is simply experience with the domain; there is no substitute for good people.

There are three main contexts for discovering scenarios:

- interviews;
- observation;
- workshops.

These all have very different effects on the project but, to quite a large extent, they are alternative ways of discovering scenarios and requirements. The techniques you will use in these contexts plainly differ because of the number of people you are dealing with (Table 5.1).

5.2.1 Interviews, storytelling

Our brains are patterned for storytelling.
Doris Lessing

In an interview, storytelling is limited by the fact that you are talking to only one or two people. Their stories may not make much sense on their own if they form small parts of larger system stories, for which workshops are better.

Table 5.1: Scenario discovery contexts compared.

Context	Advantages	Disadvantages	Techniques needed
Interviews (Chapter 11)	Full attention on one person's story	Likelihood of hearing only part of a story or business process; need to piece together evidence from different interviews	More or less open-ended questioning of the interviewee; stimulation with existing documents
Observation (Chapter 11)	Direct experience builds understanding; can discover things not easily explained	Many important scenarios are rare, and unlikely to be observed	Note taking; recording and photography with permission; apprenticing; subject gives commentary while performing a task
Workshops (Chapter 12)	Participants stimulate each other, and fill in gaps in each other's knowledge as well as yours, as you reach different people's areas of expertise; ability to identify and resolve conflicts directly; ability to obtain group consensus on requirements directly	Cost of having several people focusing on the same activity; need for skilled facilitation	Facilitation using a range of techniques, e.g. role/action list, searching for negative scenarios, etc

What you can do in an interview is to get people to talk you through the tasks that they themselves do. For example, if you are talking to an accounts clerk, they can explain the steps involved in making a payment:

'I select "Create Payments" and fill in the purchase order details over here. Then I enter the amount here . . . '

The accounts clerk may become unclear on exactly how the order is authorised:

'Then I put it on the system, marked "For Authorisation", and, um, it goes off to be authorised . . .'
 'By whom?'
 'Erm . . .'

This is what you can expect to happen when you reach the limits of an individual stakeholder's area of expertise. You can either rely on piecing together the evidence from different interviews, or you can bring the stakeholders together in a workshop.

You can use the project's mission statement and its list of objectives (see Chapter 3) to stimulate people to describe what they do:

'What do you do (in your role as quality controller) to help achieve that objective?'

And then

'Talk me through how you do that...'

You may find it helpful to explicitly give people permission to tell stories:

'Assume I know nothing about how to do that. Why don't you describe it to me as if you were teaching a new recruit how to put one of those on the system?'

5.2.2 Scenario Workshops

Workshops are ideal for discovering scenarios that involve different stakeholders, as they form a natural way to bring stakeholders together and agree their requirements.

Simple and proven techniques for capturing scenarios in workshops include:

- making role/action lists;
- making operator actions/machine actions lists;
- drawing swimlanes charts;
- asking about standard patterns (templates);
- acting scenes.

You will probably need to use a mix of several of these techniques on your project. Let's look at them in turn.

Role/Action Lists

The basic role/action technique is to make two columns on a flipchart or whiteboard, and head them 'Role' and 'Action' (Figure 5.1).

It is probably best to write the list by hand on paper, because that way you can sketch diagrams, parts of rich pictures or anything else you need, and because you get a permanent record of what you wrote or drew. You can use a computer and software such as a spreadsheet to make a list, but it is difficult

Role	Action
Cyclist	Requests display of current road speed
Device	Uses GPS signals to find its location, and the current time Uses Pythagoras' theorem to find straight-line distance between successive locations Calculates speed as distance/time interval Displays average of last 5 speed estimates

Figure 5.1: A handwritten role/action list.

to facilitate a workshop with full attention and type at the same time, and even more difficult to add drawings ‘in real time’.

A role/action list is well suited to flipchart use; it is a powerful technique, and also well suited for use in workshops. It could almost be sufficient on its own, but some variety is helpful, and other techniques can discover other aspects of a process or requirement.

When using the role/action list:

- let participants explain the scenarios to you;
- ask naïve questions! Such as:
 - ‘What do you do to<achieve result xyz>?’
 - ‘What is the first step?’
 - ‘What does “consolidate” mean here?’
 - ‘Who does this step?’
 - ‘And then what happens?’;
- deal with the normal case first – leave problems till later (see ‘Exceptions’ in Section 5.2.3). If an obvious exception pops up, just make a note of it on the side (you can circle it in red to show that it’s a special case).

Operator Actions/Machine Actions Lists

*When we design systems and applications,
we are, most essentially, designing scenarios of interaction.*

John M Carroll [5]

This is a variation on the role/action list that is especially for interactions between a single human operator and a single product. The idea is to have two columns, one for ‘operator actions’ and one for ‘product actions’ (Figure 5.2). This is suitable for designing interactions with a software-intensive product. That product may well be the human-facing part of a larger system, as for example with a bank’s teller machine, which gives you cash but also talks

<i>Operator Actions</i>	<i>Navigation Device Actions</i>
1. Asks for navigation	Offers choice of postcode, street name, grid reference, ...
2. Chooses street name	Displays A-Z list and a blank text field (as alternatives)
3. Enters street name	Checks street exists Calculates route to that street Displays start point, current direction and required direction of travel
4. Moves as indicated	Displays current position, direction, next instruction...

Figure 5.2: A handwritten operator actions/machine actions list.

to the bank to check your account, withdraw money from it and record the transaction.

Illustrating or Defining?

Be careful to show clearly whether a scenario is:

- an illustrative example, i.e. probably one of several ways of doing a task;
- the stakeholders' preferred approach (or the only acceptable approach).

You might, for example, write 'Navigation device: offers list of street names' on a flipchart: innocent in itself. But someone might later wrongly treat this as a definitive requirement. Other equally useful ways to select a location (postcode, name of city, ...) might be missed.

- **As illustrations**, scenarios can be told as colourful stories. Precision is not necessary, and only a very few cases need be covered.
- **As requirements**, scenarios should be analytic, i.e. broken down into separate steps. These must be checked for correctness. All the separate cases must be covered.

Swimlanes (One Column per Role)

The operator actions/machine actions list can be extended to multiple roles by using multiple columns, as in a swimlanes¹ diagram (in which there is a column for each role) but without necessarily drawing flowchart symbols.

¹UML provides flowcharting with its 'activity diagrams'. These can be divided with partitions into vertical lanes, horizontal lanes, or even a grid. But the use of a chart that uses direction (down or to the right) to indicate time, and lanes to indicate ownership of tasks, far pre-dates UML.

<i>Customer</i>	<i>Table console</i>	<i>Kitchen console</i>	<i>Cook</i>	<i>Train</i>	<i>Checkout</i>
<i>Chooses food</i>	<i>Displays menu</i>				
<i>Confirms order</i>	<i>Displays order</i>	<i>Sends order to kitchen and checkout</i>	<i>Displays order</i>	<i>Cooks order Puts food on train</i>	<i>Carries food to customer</i>
<i>Unloads food from train Eats the food Pays for order</i>					<i>Takes payment Prints receipt</i>

Figure 5.3: A swimlanes list for a highly automated restaurant.

Figure 5.3 uses a swimlanes list to describe the basic ‘happy day’ scenario for a restaurant that recently appeared on the news. Note that the result can be attractively clear, but that it takes up a lot of space.

Making Design Decisions without Noticing

Operator/machine scenarios drive you towards designing interactions, as you might expect. At the right time, this is a good thing; too early, and you may be prejudging various design decisions. For example:

- The project may need to evaluate the choice between design options: should there be a touch screen? A trackball? A mousepad?
- You may need to decide which and how many options to offer the operator: specify location by streetname? City? Map reference? ...

Chapter 14 explores the interplay of requirements and design.

Alternatively, you can add swimlanes to a flowchart. Where the flows are not too complicated (not too many loops and branches to worry about) this can be an effective and quick way of capturing scenarios. You may want to use a symbol (such as a red asterisk) to indicate places where branches and loops need to be thought about later, after the workshop.

Three columns are about as many as you can have on a flipchart. A whiteboard gives you more room for multiple columns but, unless your meeting room is covered with whiteboards, you will soon run out of space. As you will often need to refer back to a previously captured scenario, you shouldn't rub out what you have written to reuse the space. It is better to tape some flipchart sheets together, or to get hold of a roll of blank newsprint and stick some really large sheets to the walls.

Swimlanes in Real Time

Swimlanes and flowcharts (separately or together) are not necessarily easy to construct in 'real time' in a workshop, though you may well have them in mind for analysing and documenting your scenarios afterwards.

However, all rules are made to be broken. Perhaps you are working with a group of stakeholders, such as business managers, who are very familiar with flowcharts; perhaps you have a convenient flowcharting tool, and someone who can edit quickly enough to keep up with you while you facilitate the workshop. In that case, you may find that flowcharts and swimlanes work well for you.

Asking about Standard Patterns

Some scenario patterns recur in many products and business systems. They do not work every time, but most projects will use some of them. Common patterns that you should investigate in your scenario workshops or interviews include:

- **whole life scenario:**

- development
- bringing into service (test, installation, commissioning, etc)
- normal operations
- maintenance
- upgrade
- decommissioning/disposal;

- **a day in the life of (DILO):**

- starting up (in the morning)
- daily health check, self-test, resupply, cleaning, etc
- daily operations (e.g. transactions)
- closing down;

- **transaction/operation/task:**
 - e.g. starting the software, loading the data, processing it, displaying the results
 - e.g. navigating from A to B
 - e.g. making a conference call;
- **data handling – create-read-update-delete (CRUD):**
 - create an item (e.g. a customer record)
 - read an item (database retrieval)
 - update an item
 - delete an item (destroy a record – some systems do not allow this).

These patterns are powerful, because they lead to simple questions about essential issues. The simplest questions are often the best:

'We're reading a customer record here. Where do we create and update customer records?'

'Talk through how the system will be brought into service.'

'Why don't we step through how maintenance will work?'

If people have overlooked something fundamental, such questions should find it.

Tips for Discovering Scenarios

- Start with the normal, 'happy day' scenarios.
- Capture each story as simply as possible.
- Ask about different scopes:
 - a normal transaction
 - a day in the life of (an operator, or a product)
 - the whole life of a system or product (including manufacture, delivery, storage, installation, commissioning, operations, maintenance, decommissioning).
- Move on to special cases.
- Then look at what can go wrong, and what to do about it.
- Check for missed CRUD 'housekeeping' scenarios for handling data.

Acting Scenes

Get volunteers to act out a scenario, each person playing one role: human operator, machine, interfacing system, etc. (See Figure 5.4). No acting skill is required, just a willingness to improvise to tell a story simply and directly. This is a good way to open a scenario workshop, or to start a new session after a break. More is said about workshops as theatre in Chapter 12.



Figure 5.4: Acting out a scenario.

5.2.3 Discovering Negative Scenarios

When you have the basic ‘normal’ scenarios under control, it is time to explore what should *not* happen.

There are at least three different ways to explore negative scenarios:

- **exceptions** – events that interrupt normal progress;
- **intentional threats** – actions of hostile stakeholders, which may cause exceptions;
- **unwanted scenarios** – undesirable combinations of correct behaviours.

Let us look at each of these in turn.

Exceptions

Most negative situations, however caused, present themselves as unwanted or ‘exception’ events². An exception event is simply something that happens

²An ‘exception’ in some programming languages is a procedure for handling an internal error event of some kind. The events we are talking about here are business events, i.e. things that happen in the world, and that our system needs to handle – in software or otherwise.

(in the real world) that impedes progress towards a desired goal. The associated scenario to handle the exception is entirely positive: its job is to restore normal working if possible, or to bring about a safe and controlled stop if not.

What do we do When Something Goes Wrong?

- Exception events are special cases, important even when they are rare.
- You are unlikely to identify all the exceptions you need to handle by listing the events you expect at an interface (see Chapter 4).
- To achieve clarity and completeness, start with normal events and scenarios. Then use that structure to see where exceptions could occur.
- Your list of normal scenarios is complete when it covers all the events agreed (or seen) to be in scope in your context model.
- Your list of exception scenarios is ‘complete’ when it covers all the exception events agreed to be worth dealing with in the normal scenarios. It will never be complete in the sense of covering all possible situations.

For example, a particular railway’s signalling system is designed to assure safety by providing two independent signals to the train. These are supposed to be exactly the same. Suddenly, just outside a station, one signal tells the train to slow down to 20 km/hour; the other says to continue at full speed. What should the train do? The only allowed answer on that railway is ‘an emergency stop’. There is a general rule here: exception-handling scenarios should be simple, unambiguous, and short.

You have two choices when you have discovered a type of exception event:

1. You choose to handle it, and work out an exception-handling scenario to describe the behaviour you require for when that exception occurs. Exception-handling scenarios are a component of use cases (see Section 5.3 Documenting Scenarios).
2. You choose to ignore it, and accept the risk that the system under design will fail (and possibly cause harm such as business losses or an accident).

The dilemma is that you can’t do everything. Exception-handling typically takes up a high percentage of software development effort. A typical estimate for interactive software is 20% desired functionality, 40% user interface and 40% exception-handling, though systems vary widely.

For hardware products, physical exceptions become important, especially where safety is involved. The risks are large, and so is the degree of care required. Elaborate hierarchies of exceptions may be specified to try to

keep systems working despite multiple failures. Here is an example from a luxury car:

...

Driver signals a turn.

Car flashes front and rear turn indicator lamps.

Exception: front/rear indicator lamp failed:

Car flashes front/rear sidelight.

Car warns driver of indicator lamp failure.

Exception: sidelight lamp also failed:

Car flashes front/rear fog lamp.

Exception: fog lamp also failed:

Car flashes headlamp/reversing lamp.

...

Clearly, the underlying requirement (embodied in regulations and standards) is:

The car must indicate the driver's intention to turn.

This traces back to an assumption:

It is dangerous to turn without indicating, as other drivers will not know the driver's intention.

and to a legal argument as well:

The company would be liable to pay damages if the driver reasonably believed indicators were working when they were not.

There is a close relationship here between the exception scenarios, the requirements and the assumptions that form the rationale for the requirements (see Chapter 7).

Jackson's Principle of Commensurate Care

How much exception-handling is worth doing to prevent failures from occurring? That depends on the risk you are taking. A classic approach described by Michael Jackson (1995) [7] is to measure the Required Care (R) as the Probability of an exception (P), multiplied by the seriousness of the consequential Damage if it occurs and is not handled (D):

$$R = P \times D$$

Some serious consequences of system failure include:

- damage to reputation if products are unreliable;

- business losses, e.g. loss of income if a service is not provided;
- damage to property, injury and death, if a system causes an accident.

Avoiding Duplication in Exceptions

Exceptions have two parts: the **exception event** (the name of the exception, corresponding to an obstacle, a negative goal) and an **exception-handling scenario** to handle that event when it arises.

Often, the same exceptions appear again and again in a set of scenarios. If you make a shared list of exceptions, your team can simply cross-reference the relevant ones from their scenarios, so each exception-handling scenario is only defined once. Such cross-reference links can be handled as traces in a requirements database, or as hyperlinks in an exported specification (Figure 5.5).

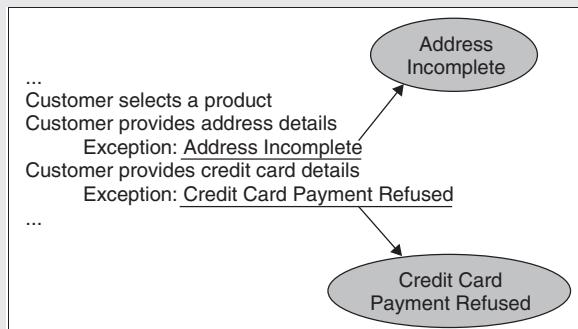


Figure 5.5: Reusing exception scenarios.

Any of these, not only the last, can be disastrous – and the required degree of care becomes enormous. Therefore, a basic approach to exception-handling is to identify P and D. If either of them are large, you should handle the exception.

In practice, two cases are common:

1. **Safety-related systems:** D is extremely large in areas such as aerospace, medical engineering, transportation, power generation and industrial automation (robotics):
 - Since D is so large, P has to be made extremely small, and the safety authorities do their best to ensure that it is.
 - Therefore, in products for such ‘safety-related’ industries, any exception that could affect safety is always treated extremely carefully, and is analysed in a safety case (see Chapter 7).

2. **Interactive products:** P is large for human-caused exceptions in interactive products (including software), especially those where the human operator is an untrained consumer. People constantly press the wrong buttons on electronic devices like music players, and forget to fill in fields on forms. In these cases, exceptions more often lead to inconvenience than harm (though serious financial loss is possible, e.g. when transmitting funds in online banking).

- Since P is large here, D has to be very small to justify not handling such exceptions gracefully.
- Therefore, in interactive software and consumer products, user interface exceptions should be handled very thoroughly. Many types of error can be prevented by good user interface design, but that is out of our scope here.

Relative Importance

A telecommunications utility makes its money by providing services such as text messaging and voice telephony. Consider the relative importance to the company of three exception scenarios:

- **Telephone switch fails:** the calls that are being handled at that moment by the switch are cut off. Users redial; maintenance operators are notified. Calls can be routed elsewhere, and the switch can be rebooted or repaired.
- **Mobile handset fails:** users immediately notice, as they cannot obtain service. If many users complain, they will overload the company's helpdesk, at a high cost in repairs, lost business and reputation. Reliable handsets are very important to users and the company.
- **Billing system fails:** users get free calls; the company immediately loses income. It is critically important to the company that billing is reliable.

Did you think the telephone switch critically important? Curiously, although switching calls is the core business of the telecommunications utility, the temporary loss of a switch may not be the most critical kind of failure.

Intentional Threats

You may also need to look for intentional threats from hostile stakeholders (see Chapter 2), also called 'misuse cases' (Alexander and Maiden 2004) [2]. Then, explore the negative goals (see Chapter 3) that those stakeholders could

have. The principle of commensurate care ($R = P \times D$) applies: you should care about threats that are serious and likely to happen.

The responses to these may require additional exception-handling scenarios, or may bring other requirements (e.g. business rules, security requirements) into being.

Scenarios = Acceptance Test Cases

In general, system tests of products are not sufficient. They need to be supplemented by thorough testing of realistic scenarios. These should include scenarios that stress the whole system (product, people and external interfaces). Such scenarios often overload systems in unexpected ways.

Test engineers are skilled at devising troublesome scenarios of this kind. Classics include:

- **Peak load scenario (= stress test):** run the system up to (say) 110% capacity briefly, and see if it falls over.
- **Continuous load scenario (= soak test):** run the system at 100% capacity for N hours.
- **Failure injection scenario:** make something fail (pull a card out of a computer, reboot a server, unplug a network cable, etc) and see if the rest of the system can keep going.
- **Day in the life of (DILO) scenario:** play right through the whole of a normal day of operations, including logging on, taking meal breaks and so on.

Note that all the scenarios you discover – positive and negative – are likely candidates for test cases. The earlier you talk these through with your test people, the better.

The basic way to discover threat-handling requirements is:

1. Identify hostile stakeholders.
2. Identify their (negative, threatening) goals by asking, ‘What can this stakeholder do to threaten our goals here?’
3. Identify the needed responses, by:
 - a. asking, ‘What goals should we have to counter those negative goals?’
 - b. working out how to achieve those threat-handling goals, either by goal analysis or with scenarios.

For example:

1. Hostile stakeholder: Hacker

2. Threat: Hacker breaks into system, steals or corrupts data
3. Responses: Forbid remote login; enforce strong passwords.

Negative scenarios are often very suitable for acting out as scenes (see ‘Acting Scenes’ in Section 5.2.2). For example, one person could play the hacker trying to break into the accounts system, another person plays the accounts clerk, another the security manager, etc. This directs people’s attention more effectively than simply asking them what their security goals are.

Unwanted Scenarios Composed of Correct Behaviours

A third use of negative scenarios is to identify combinations of correct behaviours that you require *not* to happen.

For example, if you are at a filling station, you need to know how to fit the petrol cap back on to your car when the tank is full; that’s one correct behaviour. You also need to know how to make fuel flow from the pump. But the scenario composed of these two correct behaviours in sequence:

‘Fit the petrol cap. Make fuel flow from the pump.’

is negative – it is wasteful and dangerous.

The response could be to require the pump to check that the nozzle is safely inside (or locked on to) a fuel pipe before turning on the fuel flow.

The discovery of a negative scenario should thus lead to additional requirements. These are typically to improve safety, security or reliability; i.e. the additional requirements will initially be for *qualities* (see Chapter 6). Those qualities will often be provided by adding specific functions (such as checking the state of the fuel nozzle).

Chapter 3 describes how to discover obstacles and threats (i.e. negative goals); these, in turn, may need to be treated as *exceptions* (from the point of view of your product) and accordingly handled with scenarios.

In a scenario workshop, you can simply point at a step in a scenario that you have captured on a flipchart, and ask:

‘What can go wrong here?’

This question is powerful enough for many purposes. If they are familiar with the domain, workshop participants will quickly suggest the main problems that can arise.

To discover exceptions (whether or not you are creating use cases, though the two go together well), have a separate activity to go over the scenarios you have documented on your flipcharts. Ask questions like:

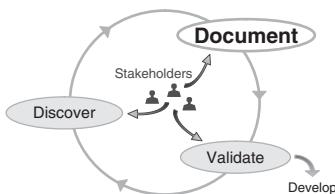
- Could the operator make a slip here?
- What would happen if the operator failed to notice this?

- What if this message was lost/delivered out of order?
- What if this product failed here?

Tips for Discovering Negative Scenarios

- Use the structure of your scenarios to drive discovery of possible problems. Look at each step in turn.
- If the business already knows the types of exception that can occur, put them into a checklist and use that to help drive discovery.
- Consider using a list of generic types of exception (e.g. human error, machine breakdown and loss of communications) to help drive discovery.
- Be realistic; you can't cover everything.
- Use the principle of commensurate care ($R = P \times D$).
- Record only enough detail to make the negative scenarios clear.
- Avoid duplication; keep a list of known exceptions, and refer to it.

5.3 Documenting Scenarios



It is important to choose the right amount of detail for your scenarios. The different approaches that we will describe in detail are:

- **Unstructured stories:** brief, informal, possibly told as examples with incidental and even humorous detail, and quick to document. These may be sufficient when operational stakeholders and programmers are in close contact, as in an agile software development team.
- **Storyboards:** time-sequences of simple pictures or diagrams, to tell stories graphically. The technique came from film, where many people have to collaborate to get many details right for each shot to ensure continuity. Storyboards have obvious advantages for working with stakeholder groups.

- **Operational scenarios:** relatively formal engineering stories, possibly structured into numbered steps in time-sequence, describing intended operations or the use of a product, for a typical case, without incidental detail.
- **Use cases:** highly structured stories, each consisting of a set of (operational) scenarios to achieve a stated goal under different conditions, together with associated requirements.

Use cases are time-consuming and costly to construct, but are effective in giving detailed guidance to programmers and testers. This may be necessary where you are contracting out the implementation of your software to programmers who are not familiar with your domain.

Table 5.2 summarises the trade-offs between these approaches. They are certainly complementary to each other.

Tips for Documenting Scenarios

- Decide what style will be most appropriate (colourful stories or something more analytic).
- Tell your story simply and directly.
- Supply enough context to enable readers to see where the scenario fits in.

5.3.1 Index Cards, User Stories

All we know is embodied in stories. We understand everything in terms of stories we already know.

Roger C. Schank, Tell Me A Story

In the agile style of software development, scenarios in the form of ‘user stories’ (Cohn 2004) [4] are written on index cards. Each story’s card is used to track the story as it is implemented in software, tested, and accepted by the customer. The technique is simple and powerful, and is ideal when formal (contractual) requirements are not needed. This may happen if you are developing software in-house (for your colleagues to use) or if you are working closely and informally with your customer.

‘Index card’ is also a powerful metaphor. Each story can be held in a database record. The record can be displayed on screen as an ‘index card’. The story’s status can then be tracked using attributes for owner, priority, status and so on.

Table 5.2: Choosing the right scenario style for your project.

Style	Structure	Advantages	Disadvantages
Stories	One index card per story	Short, cheap, quick, easy to write; clearly illustrative, so not giving false impression of complete coverage; good for 'agile' (iterative) development	Can't trace to individual steps, so it is hard to monitor progress below the level of a whole story
Storyboards	Frames drawn singly or in a sequence like a cartoon strip	Effective in workshops, and to stimulate creativity; easy to understand, good for a wide range of non-technical stakeholders, good lead-in to user interface design	Informal
Operational scenarios	Story, i.e. a time sequence, possibly as numbered steps	Easy to construct in workshops, can trace to individual steps	Not as systematic as use cases; traditionally not used as requirements
Use cases	Goal, main story, variations, exceptions, preconditions, guarantees, NFRs	Fully analytic, designed to cover all scenarios including variations, exceptions; also covers preconditions, guarantees; can drive development across contractual boundaries	Time consuming, costly to develop; give impression of completeness but provide poor coverage of NFRs in particular (see Chapter 15)

5.3.2 Storyboards

A storyboard represents the steps of a scenario as a sequence of pictures, diagrams or successive imagined 'screens' of a product. A storyboard is effectively a paper prototype (see Chapter 13).

Storyboards can take many forms. They can illustrate positive or negative scenarios, whether or not software is involved.

For example, Figure 5.6 tells the story of a young mother with a pushchair, faced with a poorly designed access route to the trains in a railway station. A good outcome that such a storyboard can trigger is the adding of requirements

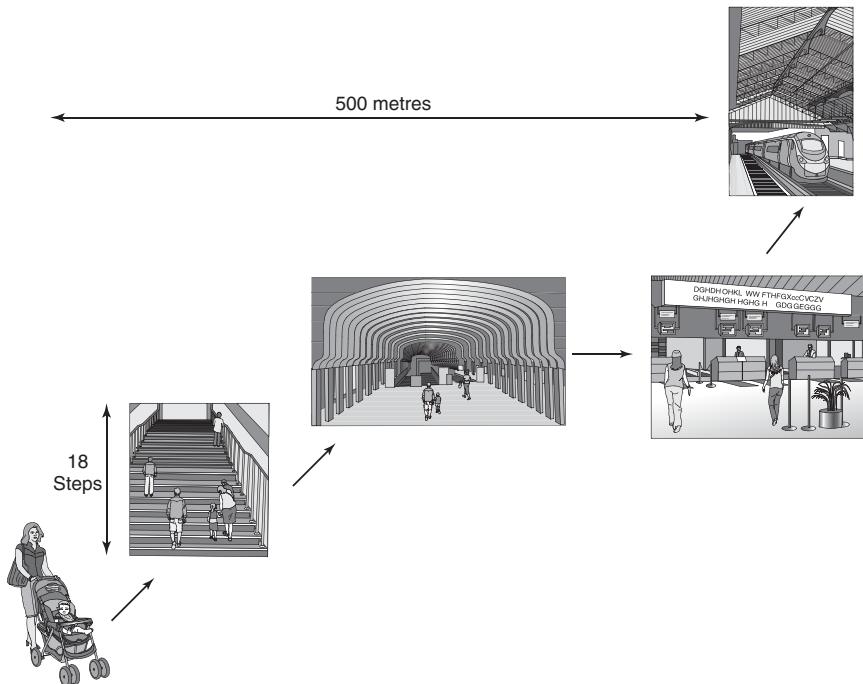


Figure 5.6: Storyboard illustrating a negative scenario.

(in this case for accessibility for those with baggage, children or disabilities) and the subsequent design of a good, workable system. It does this by revealing unconsidered assumptions (see Chapter 7) about how easily the public can climb stairs, walk with baggage and so on. Once identified, these bad assumptions can be replaced with agreed assumptions that will effectively become constraints, e.g. that passengers will not all be able to (and hence, must not have to) walk more than so many metres without access to baggage trolleys, etc.

Storyboards can be used to give a general idea of an issue. They are equally good for detailed use in a workshop, to build an agreed approach to a problem.

For instance, suppose you are creating a multifunction device (MFD) for outdoor leisure use. You hold a scenarios workshop. People suggest ideas, and together you create some storyboards on flipchart paper about how the device could be used by fishermen, cyclists, in the mountains and so on (Figure 5.7).

You do not need to be an artist to draw useful storyboards, though if you have someone in the team who can draw well, that may help. You can use any materials that come to hand – photographs, clipart (though that can definitely be overused), diagrams, sketches. It is often much quicker to draw with a pen and black ink on white paper, than to spend hours with a computer and a graphics package giving unnecessary polish to storyboards.

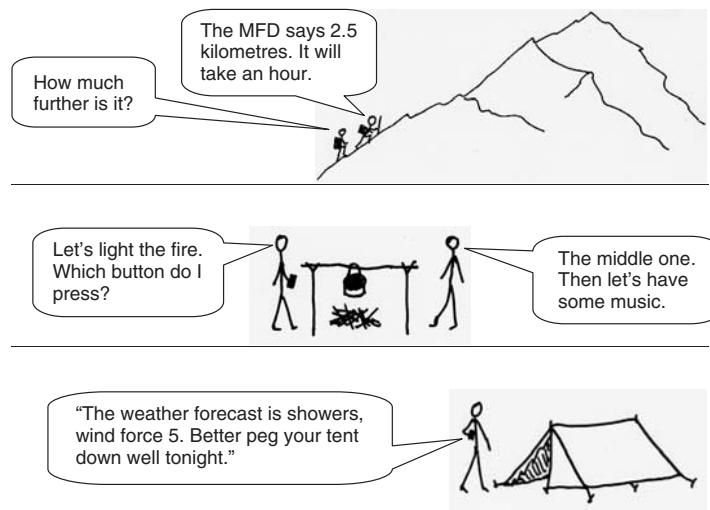


Figure 5.7: Hand-drawn storyboards.

Tips for Storyboarding

- Keep it simple; focus on the essential message.
- Scan pen-and-ink sketches rather than wasting time with graphics tools.
- If your handwriting is hard to read, use a computer to create speech bubbles and titles for hand-drawn images.
- Bring a digital camera to workshops to capture large images from flipcharts.

5.3.3 Operational Scenarios

An operational scenario is an engineering story that describes the operations that the product or service is intended to support. The real value of this is to show developers, who may be far from actual operations, what result they should be trying to achieve, rather than a set of disconnected features.

For example, when buying airline tickets, passengers often want to compare times and prices without having to re-enter all their data just to view an alternative flight time:

1. Customer specifies origin, destination, number of passengers and approximate date/time of flights out and back.
2. Airline offers choice of flight times and classes of ticket at different prices.

3. Customer selects desired flights, identifies the passengers and pays for the flights.

Caution: Interaction Storyboards Lead to User Interface Design

If you use storyboards of a mocked-up user interface, be aware of the risk of diving prematurely into user interface design.

If you are still early in the requirements phase, it may be best to keep the design very rough, e.g. with hand-drawn ‘screens’ (see Chapter 13), screen fields and buttons as hand-labelled sticky notes, etc. You might also want to emphasise that the purpose of the workshop is to agree the requirements, not to design the user interface in detail.

Alternatively, you might decide to stay with written scenarios until the project is ready to move further into user interface design.

Notice that the operation is described without focusing on the mechanisms of the product or service. It could be a web browser, a kiosk, a travel agent or a booking desk at an airport.

Operational scenarios have traditionally been used to give context to lists of requirements, but not as a means of actually stating requirements. The aerospace and defence industries distinguish different levels, including:

- the higher-level **concept of operations** (what results should be produced in the world, possibly involving many people and products working together);
- the lower-level **concept of use** (how interaction with a product should work). This implies some knowledge of the chosen design, and may therefore be more appropriate after optioneering (see Chapter 14).

Operational scenarios can be told in any way that is convenient, such as a text with numbered steps, a role/action list (Figure 5.1), an operator/machine actions list (Figure 5.2), a swimlanes chart (Figure 5.3), even a flowchart.

You can use these techniques whatever level you are working at, from a whole business down to interactions with an individual piece of equipment. When this amount of detail proves insufficient, you should switch to use cases.

5.3.4 Use Cases

A use case is a structure for describing a group of related scenarios that are meant to achieve a named functional goal. It can take up to 10 days’ work to create a fully-documented use case, so:

- People often document use cases only sketchily; this may be the right thing to do, especially at the start of a project. For instance, you might do

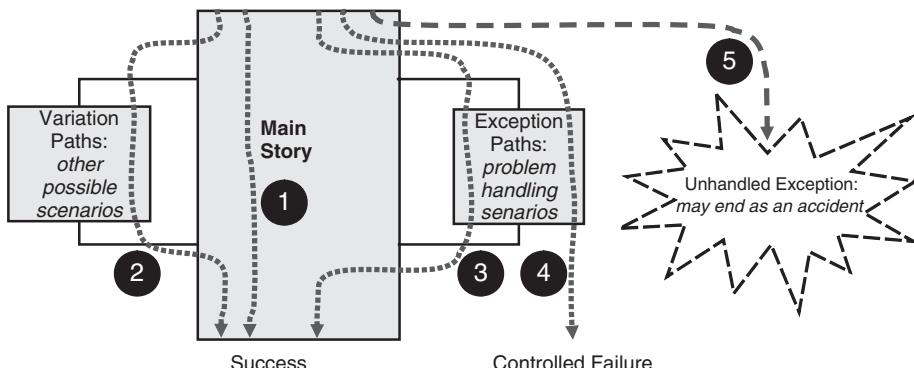


Figure 5.8: The structure and behaviour of a use case.

best to wait until you know that a use case is in scope before defining it in complete detail.

- You should not immediately assume you have to create use cases just because you have some scenarios to write down.

There is broad but not perfect agreement in industry on the pattern or template for use cases. The structure described here is based on what Alistair Cockburn (2001) suggests as a ‘fully dressed’ use case in his helpful book *Writing Effective Use Cases* [3], modified slightly to avoid software-specific language.

The key idea of a use case is to describe ways that someone can use a product to achieve a desired result. These scenarios are shown as lines on Figure 5.8:

- Scenario ①: the normal way of achieving the goal;
- Scenarios ②: other possible ways of achieving the same goal;
- Scenarios ③: recovering from exception events to achieve success;
- Scenarios ④: handling unrecoverable exceptions safely;
- Scenarios ⑤: the exceptions you choose not to handle.

Once you understand this is how a use case works, you will see that the structure, set out as a template in Table 5.3, is actually quite logical, and as simple as it can be to do its job. If you don’t need so much detail, you should concentrate on the goal and main scenario first, and omit the other parts.

It is possible and fashionable to try to force all scenarios, and indeed all requirements, into the shape of use cases. However:

- At best, this is often a waste of resources, and simpler scenarios could well be enough. Not every project needs that much detail.
- At worst, it can cause serious delay. That can make people feel that requirements are a waste of time. They might then cut corners, e.g. starting development based on unstated assumptions and incomplete requirements.

Table 5.3: 'Fully dressed' use case template.**Use case:**

- **Functional goal (= use case title)**
- (stated as an active verb phrase, e.g. 'Navigate to destination');
- **Primary and other roles** (called 'Actors' in UML);
- **Main/normal/happy day scenario:**
 - Step 1
 - Step 2, etc;
- **Variation scenarios** (maybe branching off from and maybe returning to the main scenario; similarly narrated in steps):
 - Variation 1:
 - Variation Scenario 1
 - Variation 2, etc;
- **Exceptions**, i.e. exception events and their exception-handling scenarios (branching off from any scenarios in the use case and, if possible, returning to them, or else ending with failure; again narrated in steps):
 - Exception event 1:
 - Exception scenario 1
 - Exception event 2, etc;
- **Preconditions**, including the **trigger** condition that causes the use case to start;
- **Guarantees (postconditions)** (what must be true when the use case ends):
 - the success guarantee(s) - what must be true for the use case to end with success
 - minimal or failure guarantee(s) - what must be true, however the use case ends;
- **Stakeholders and interests:** people/organisations who have valid interests, briefly stated here, in this use case;
- **Local qualities and constraints** (nonfunctional requirements) that apply specifically to this use case (rather than 'globally' to the whole product; shared information should be held in a central place and linked to the use cases that need it to avoid duplication.)

- A 'fully dressed' set of Cockburn use cases is necessary when:
 - the scenarios are to serve as the functional specifications
 - the contract is to be costed against the scenarios, which must therefore be 'complete' (i.e. any changes will be paid for separately).
- In other life cycles (e.g. where development is in-house or software is being coded by members of an integrated project team), outline use cases (goal, main scenario, postconditions) may be sufficient.

Table 5.4: Approximate effort to prepare use cases.

Amount of detail in the use case	Effort per use case (man-days)
Sketch (goal, role, brief description only)	0.1 to 0.5
Outline (goal, role, main scenario)	1 or 2
Fully dressed (complete template)	10

Tips for Documenting Use Cases

- Start with titles (use case goals) only.
- Make each title a function, something to do; write it as a verb phrase (a predicate) in the general style, 'Issue an invoice'.
- If you can put 'I want to' before the title, it is probably correctly structured.
- Write the main story as a simple list of actions.
- Write the guarantees, i.e. what should be true at the end.
- Consider whether any more detail is needed; if it is not, stop.
- List the important exceptions.
- Only then, fill in any other details.

As a *very rough* guideline, Table 5.4 indicates typical effort on use cases. This take into account the work needed to discover, draft, validate and review the scenarios.

Organising Use Cases

Use cases were invented to make the job of organising requirements easier. It is ironic that use cases are themselves quite difficult to organise.

In principle, individual steps should be organised into scenarios, and related scenarios should be grouped into use cases. A good use case makes sense as an end-to-end story or transaction from the point of view of the people involved; it achieves something outside the product or system to which it relates.

Inside-out Use Cases

The wrong way to identify a use case is to observe that you have a set of functions or user interface controls, and to translate these into supposed use case titles: ‘Switch on word-wrapping’; ‘Disable overwrite mode’.

These are bad because it is unlikely that any user comes to your software with the intention of achieving such a thing in a work session, and because they are too small to be worth the overhead of an entire use case in your documentation. They are also ‘inside out’; a use case should lead to, justify and organise many such small functions, rather than the reverse.

Equally, a use case is not a software object. A good software use case describes behaviour that traverses many (future) objects, to achieve an end-to-end result outside the software. More than one use case may visit a particular software function, too.

CRUD scenarios for data handling (see ‘Asking about Standard Patterns’ in Section 5.2.2) are often problematic, as Cockburn (2001) explains [3]:

- Logically, there are four separate use cases ('Create an XYZ', 'Read an XYZ', etc) per data item. Unfortunately, this implies a proliferation of often trivial use cases.
- To keep the number of use cases down, you can write a single if somewhat artificial CRUD use case ('Manage an XYZ'), using variant paths for the different scenarios. Unfortunately, this violates the spirit of the use case concept, which is to name and describe a single end-to-end action that people want to use a product to achieve.

Other possible organising principles include:

- Group use cases by their primary role, e.g. a set of use cases for the retail sales assistant, a set of use cases for the sales manager, etc.
- Group use cases by their operational context, e.g. a set of use cases for sailing the ship, a set of use cases for loading/unloading operations in port, a set of use cases for maintenance in dry dock, etc.

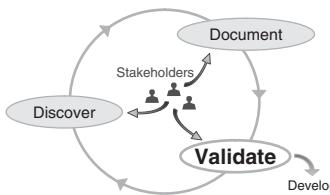
It is helpful to draw a separate diagram for each such group of use cases, to act as a graphical index. Roles are drawn as stickmen, and use cases are then drawn (with absolutely no internal detail) as elliptical bubbles. You can then provide a hyperlink from each bubble to the use case definition.

5.4 Summary

Scenarios are immensely valuable for describing business requirements. They are so powerful that people sometimes think that they make all the other requirement elements unnecessary. We will say more about this in Chapter 15.

Not surprisingly, people have invented many different ways of documenting scenarios, including stories, storyboards, operational scenarios and use cases. Whichever you choose, remember Alistair Cockburn's advice (2001): 'Find a way to make the story shine through'. [3]

5.5 Validating Scenarios



Scenarios are the basic way for operational stakeholders to describe how they work and how they want to work in future. The key to validating scenarios is to ensure that the same stakeholders are happy with what has been documented.

Scenarios provide a powerful means of checking for completeness, correctness and consistency, by virtue of their story structure.

There should be scenarios to cover all functional goals.

Tips for Validating Scenarios

- Read them through. Do they make sense?
- Have any steps been left out?
- Is each step a simple, 'somebody does something' action?
- Above all, check with the scenario owners, the stakeholders.

5.5.1 Scenario Walkthroughs

A walkthrough is a structured workshop with the single purpose of finding and fixing errors in a set of scenarios.

The main kinds of error in scenarios are:

- steps missed out;
- steps in the wrong order;

- steps wrongly shown in a fixed order, when they could be done in parallel (which could be quicker, or easier if it means synchronisation is not needed):
 - for example, a racing car has all its wheels changed and is refuelled simultaneously by the crew during its brief pit stop in a race
 - you can express this by saying ‘The following steps may be conducted in parallel’ and then providing a sub-list with a different numbering style, say (a), (b), (c) rather than (1), (2), (3);
- variations overlooked, e.g. by trying to cover different situations with a single scenario;
- roles and responsibilities wrong, e.g. claiming that a task must be done by a manager when it can be done by a clerk, or vice versa;
- exceptions overlooked;
- coverage uneven, e.g. the steps are a mix of vague and detailed, or there are too many scenarios on normal operations but none on maintenance, etc.

Tips for Validating Use Cases

Use the Tips for Validating Scenarios from section 5.4 on each scenario in the use case.

Also check the following points:

- Is the use case title a functional goal (to do something)?
- Is the primary role (actor) identified?
- Is the main story told simply and definitely (no ifs and buts)?
- What can go wrong? Are all the important exceptions covered?
- Are the necessary preconditions stated?
- Are the success and failure guarantees defined correctly?
- Are any necessary constraints and qualities stated?

The basic walkthrough approach is as the name implies: you step through the scenarios one at a time. This sounds laborious, but with the right people present in the workshop it is very productive in detecting errors and identifying missed requirements.

You can use a quite neutral and general facilitation approach, asking:

- ‘Are these steps in the right order?’

- ‘Are the roles and responsibilities here correct?’
- ‘Have we missed any exceptions here?’

and so on.

Simple corrections to the documented scenarios can be made immediately by a ‘scribe’ and projected for everyone to check. The facilitator needs to be aware of the risk of getting the whole workshop into ‘edit the punctuation’ mode. The key is to ensure that comments are at the level of *what the product needs to do*, not how the document is formatted.

More complex corrections should be recorded as actions to be completed after the workshop.

5.5.2 Animation, Simulation, Prototyping

One of the best ways to validate a scenario of any kind is to play it through in some appropriate way to a suitably qualified team. As you go through the scenario, the team flags up missing or misplaced steps, and identifies additional requirements, missed exception scenarios, and so on.

You can play through a scenario by:

- demonstrating it on a real system if one exists, perhaps an earlier version of a product, or a prototype;
- acting it out as a scene, with or without a mock-up or drawings of the product;
- showing a film, animation or simulation of what the scenario would be like;
- stepping through a series of drawings, sketches, diagrams, screenshots or anything else that enables you to represent each step of the scenario in turn.

Tips for Playing Through a Scenario

- Do it boldly; play is an extremely effective way of learning. If you look embarrassed, everyone else will be too. Show that playing through a scenario is fun but serious, and the team will join in and work hard.
- Choose a technique that you feel comfortable with.
- Prepare carefully.
- Plan for how you’ll record and incorporate comments. You may need several people to ‘scribe’ and facilitate.

5.5.3 Things To Check Scenarios Against

- Functional goals (Chapter 3)
- Events from context model (Chapter 4)
- Situations mentioned in interviews (Chapter 11) and workshops (Chapter 12)

Validated Features?

Relaxing with a beer at a trade show, some salesmen from a software product company boasted that they had some new training courses that were structured around scenarios that the product's users most commonly asked about: 'It makes more sense, and it's much easier to demonstrate.'

There were, they said, many other features of the product that didn't fit into any known scenario. 'But our marketing chief likes them!' said one fellow merrily, raising his glass.

'It makes me wonder why the features are in the product at all,' said one of his colleagues.

Perhaps the marketing chief's features had slipped through development without any demonstrated goals (Chapter 3), context (Chapter 4), scenarios (this chapter), rationale (Chapter 7) or, indeed, agreed priority (Chapter 10). Perhaps they weren't requirements at all.

5.6 The Bare Minimum of Scenarios

- Agree how the product ought to work for its users.
- Identify the show-stoppers among the things that might go wrong, and handle them.

5.7 Next Steps

- Once you have a good understanding of your system's wanted behaviour, turn your attention to the qualities and constraints (Chapter 6) that it must have.
- Continue to improve the project dictionary (Chapter 8).
- Prioritise your scenarios (Chapter 10).

5.8 Exercises

1. You are working on a project to produce a commercial, multi-functional device for outdoor leisure activities. The basic goal is to make money by selling the device to people pursuing a wide range of outdoor hobbies and sports. The marketing concept is for an easily carried device that 'does everything' and costs 'about the same as a camera', helping with navigation, entertainment, safety, and practical tasks.
For example, a climber could use the device to display her exact altitude, and hence work out how far up a climbing route she was.
Draft scenarios to show how the device could be used by:
 - a. mountain walkers;
 - b. canoeists;
 - c. fishermen;
 - d. campers;
 - e. cyclists.
2. Take one of your scenarios from question 1 and document it as a use case with title (functional goal), exceptions and guarantees.

5.9 Further Reading

5.9.1 Storytelling

1. Schank, R.C. (1990) *Tell Me a Story: Narrative and Intelligence*, Evanston, Illinois: Northwestern University Press.
Schank, a pioneer of artificial intelligence, became interested in the power of stories to structure knowledge and communicate it effectively. If intelligence and expertise are largely about making useful connections, it may be that the connecting thread of story is a primary element in how humans think.

5.9.2 Alternative Scenario Approaches

2. Alexander, I. and Maiden, N. (Editors) (2004) *Scenarios, Stories, Use Cases*, Chichester: John Wiley & Sons, Ltd.
Alexander and Maiden have put together a detailed set of examples of how scenarios can be used in requirements work, both from research and in industry. The examples are tied together in a standardised chapter

structure to make comparison easier. The editors also introduce the field and summarise the book's findings. There is a chapter on negative scenarios and misuse cases.

3. Cockburn, A. (2001) *Writing Effective Use Cases*, Boston: Addison-Wesley. Cockburn took the minimal suggestions made in UML and developed them into a robust and justly popular structure for documenting complicated software scenarios. His book is clear and very well illustrated with real examples of a wide range of use case challenges.
4. Cohn, M. (2004) *User Stories Applied: For Agile Software Development*, Boston: Addison-Wesley. Cohn argues forcefully for rapid iteration, using brief stories instead of heavily formalised requirements. Whether or not your project is suitable for such an 'agile' approach, there is much to learn from Cohn's efficient handling of stories.
5. Carroll, J.M. (Ed), (1995) *Scenario-Based Design: Envisioning Work and Technology in System Development*, New York: John Wiley & Sons, Inc. Carroll has edited a wide-ranging, interesting selection of scenario approaches with a strong flavour of human interaction design. One of the chapters is an early cameo appearance by Ivar Jacobson on use cases.

5.9.3 Running Scenario Workshops

6. Gottesdiener, E. (2002) *Requirements by Collaboration: Workshops for Defining Needs*, Boston: Addison-Wesley.
See the Further Reading section of Chapter 12 for comments.

5.9.4 The Principle of Commensurate Care

7. Jackson, M. (1995) The Fudge Factor. In *Software Requirements and Specifications*, Harlow: Addison-Wesley.
Jackson, as always, is terse and right to the point in his essay on doing things properly.

CHAPTER SIX

Qualities and Constraints

Quality cannot be retrofit into software.

Alan M Davis

Requirement Elements	Qualities and Constraints
Priorities	
Measurements	
Definitions	
Rationale and Assumptions	
Scenarios	
Context, Interfaces, Scope	
Goals	
Trade-Offs	
Stakeholders	
Discovery Contexts	
Introduction	
From Individuals	
From Groups	
From Things	
Trade-Offs	
Putting it all Together	

Answering the questions:

- How do you want this product to be?
- What qualities would make this a really good product?
- What constraints are there on the product?
 - ... so requirements other than for product behaviour are not forgotten
 - ... so the product meets people's expectations
 - ... so the product works in its real environment.

6.1 Summary

Qualities and constraints (known together as non-functional requirements or NFRs) show that you know how people expect the system to be, rather than what they want it to do. These are critically important requirements.

Qualities such as safety, performance, reliability, usability and durability (hence collectively called '-ilities' by engineers) govern consumers' product choices and company reputations, in a world where functionality is often much the same across many manufacturers' offerings.

Constraints such as compliance with standards, or the maximum weight or power consumption of a device, may form a large part of a specification. They may be unexciting but they must not be ignored.

6.2 What are Qualities and Constraints?

6.2.1 A Rich Mixture

The name 'nonfunctional requirements' says it all; this is a huge mix of often poorly understood needs, including every type of requirement except one: functions. Perhaps it is natural that people should start by working on what a system has to do. This is especially true for software, where functionality is what is new about it. The NFRs have been left in second place, the Cinderella of requirements, sitting neglected at home while her fine sisters Functionella and Use-Casella went out and had a ball.

6.2.2 Qualities that Govern Choices

Of course it is safe: we certified it.

An FAA Administrator

Think about how you choose a car. Are you really interested in the finer details of functionality, say that the car has a liquid crystal temperature display

configurable to show Celsius or Fahrenheit? Or are you more concerned with the essential qualities of the car - its safety, comfort, performance, reliability and running costs? It's no contest.

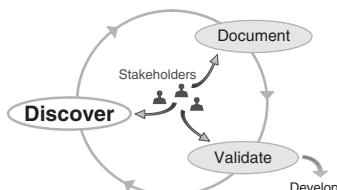
Qualities distinguish a good product from a bad one. This is because qualities are difficult and expensive to build in. You can add a function in a few days or even perhaps in a few hours, but qualities can't be sprayed on at the last minute. Products with many quality requirements have to be planned deliberately, in advance. Software projects that have few quality requirements can be run in a much lighter, more 'agile' way: but that is not to say that every project can be agile.

Qualities depend on all parts of a system working together, which is why people sometimes say they 'emerge' as properties of systems. The truth is rather that qualities have to be designed in from the start. Every part of a product (and all of its functionality) has to be carefully arranged to help to deliver the required qualities.

6.2.3 Constraints that Matter to People

Constraints are requirements that impose explicitly defined limits on how products are to be constructed. There are many kinds of constraints, such as on size, weight, power consumption and compliance with standards. The one thing that all of these have in common is that they matter tremendously both to customers and to operational stakeholders.

6.3 Discovering Qualities and Constraints



There is a tension in work to discover qualities and constraints between analysis, which often needs to be done by skilled individuals (such as safety or security specialists), and the need for such important, costly and pervasive requirements to be considered and agreed by the whole team.

You may need to think out a suitable balance between working with individuals (Chapter 11) and working with groups (Chapter 12) to discover these nonfunctional requirements on your project.

You will probably, in any case, need to prioritise (Chapter 10) and trade-off (Chapter 14) your quality requirements with your project team, even if discovery is mainly the work of specialist individuals.

6.3.1 Using Goals to Discover Qualities and Constraints

The most direct route to qualities and constraints is from the goals of the stakeholders concerned. Goals (Chapter 3) are of many kinds, and inevitably include the key qualities and constraints that are on stakeholders' minds. Remember that a goal is something that a stakeholder wants, but that it may not be expressed in a measurable, and hence testable, form.

For example, suppose you are running a workshop to discover people's quality and constraint goals for a multi-function device (MFD) to support outdoor leisure activities such as canoeing, hiking, climbing and camping. The workshop may come up with a range of different kinds of goal, very likely at differing levels of detail (Figure 6.1).

When you look at a workshop's outputs, you'll immediately notice that people's concerns are as varied as their backgrounds. Issues of dependability and usability appear alongside concern about the device's weight and whether they will lose it if it is not tied on securely. Some of these will turn out to be critical to the project; others will just be the latest fad or fashion, and may not matter at all – beyond your acknowledging that all stakeholders deserve to be taken seriously.

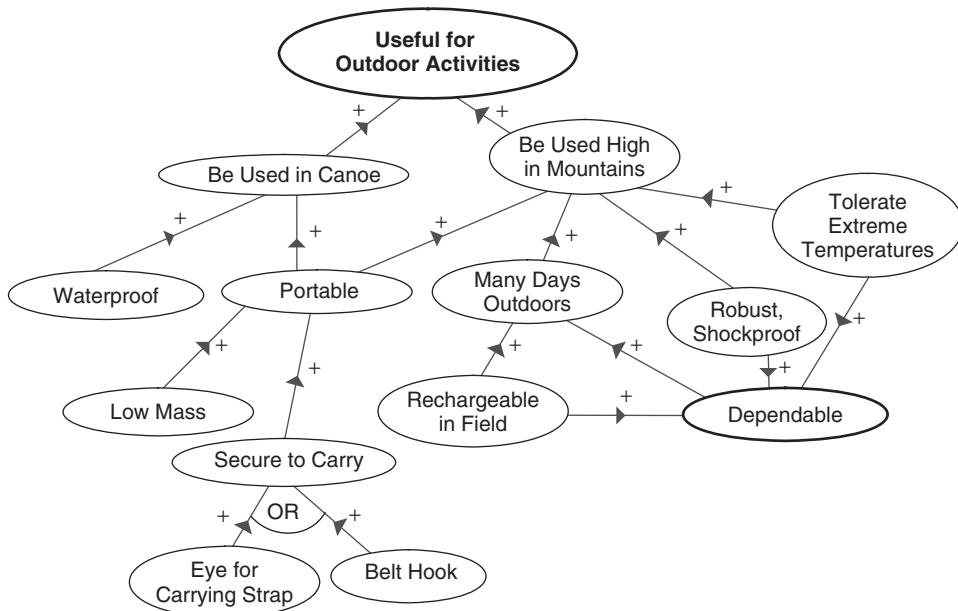


Figure 6.1: Nonfunctional goals for a multi-function device.

Let us look at just one of these ‘nonfunctional’ goals in a little more detail. Being dependable (as shown in Figure 6.1) is evidently a critical concern for an outdoor activity device. If you get your workshop to analyse the basic dependability goal, you might find that it can be analysed further (Figure 6.2), yielding new candidate requirements.

Some of the goals that you discover may be alternatives or partial alternatives. These can be shown on a goal diagram as ‘OR’ and ‘AND/OR’ respectively, inside arcs, as shown in Figure 6.2. For example, you could demand long battery life, or the ability to recharge the battery in the field, or both. Both of those approaches have costs and benefits (see Chapter 14). For instance, longer battery life can be provided with larger or more advanced batteries, at a price in additional mass and component cost; or by more advanced power management, at a cost in software development and special hardware (e.g. processors that can be partially shut down to save power).

Figure 6.2 expands and reorganises part of Figure 6.1 to emphasise dependability. It also suggests that a common quality like dependability can be achieved in widely varying ways, according to the product. In other products, functional features like rubber armour and a wind-up charger might be completely inappropriate. Dependability is considered in more detail later in this chapter.

The top-level goal of Figure 6.2 is broken down into major sub-goals that are still nonfunctional in character. But the next level down starts to reveal specific system functions, such as the ability to recharge the battery by winding a crank. Such functions may combine the use of noncomputational hardware

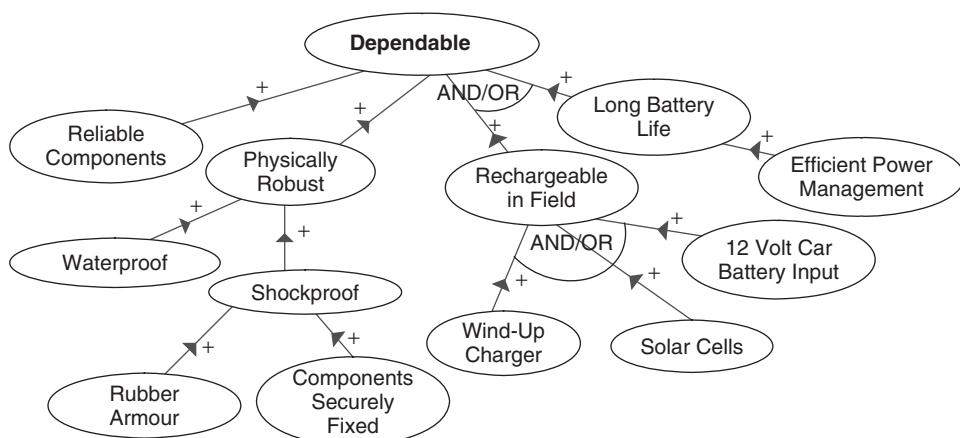


Figure 6.2: Analysing a dependability goal.

(like the crank) with computational hardware and software, for example for efficient power management to prolong battery life.

Two general rules can be inferred from this example:

1. Quality goals (like being dependable) often break down into functional requirements, for hardware (like wind-up charging) or software (like power management). As another example, high-level security goals quickly break down into software functions, such as filtering email, detecting viruses, encrypting messages and handling passwords.
2. As high-level goals are broken down, they become more specific and easier to measure.

These rules are useful, as they tell you what to do when faced with a vague high-level usability goal like 'be easy to use'. As that stands, it is not verifiable – you have no idea what to measure – so you can't use it as a requirement. What you need to do is to find a way to measure it, or to decompose such goals into smaller parts, until you arrive at requirements that you can use to drive design and verification activities, without losing sight of the original goal. The section on Usability in Section 6.4 suggests some ways to do this.

6.3.2 Stakeholder Analysis to Discover Qualities and Constraints

Another way into an understanding of required qualities and constraints is from your analysis of your project's stakeholders. Table 2.4 in Chapter 2 indicates the principal sources of different types of requirement. For example, if your project is creating a hardware device with embedded software, that table reminds you that you should meet the hardware manufacturer to discover any producibility requirements.

If you customise a stakeholder model for your own project, you will be able to create a more specific list of the kinds of requirement that you expect from each of your stakeholders. You can then use that list to help ensure you have a complete set of qualities and constraints, for example when holding interviews or workshops with different groups of stakeholders.

6.3.3 Using a Checklist to Discover Qualities and Constraints

A Hardware + Software System Checklist

System projects differ markedly from each other; for instance, projects to develop a handheld device, a web service, a jet engine, a new urban railway and a banking network will have very different emphases and types of requirement, even if all of them are controlled by software.

- Constraints
 - Design Constraints
 - Use of COTS Products
 - Forbidden (e.g. toxic) Materials
 - Construction
 - Disposability (for recycling)
 - Regulations & Standards
 - Human Factors
 - Environmental (only a few example subtypes are shown)
 - Temperature Range
 - Humidity, Waterproofing, Corrosion Resistance
 - Vibration, Shockproofing
 - Electromagnetic Compatibility (EMC)
 - Allowed Emissions (not to harm other systems)
 - Required Immunity (from other systems' emissions)
 - Physical
 - Size & Shape
 - Weight
 - Power Consumption
 - Finish, Colour, & Labelling
- Development Qualities
 - Producibility (Manufacturability)
 - Flexibility
 - Upgradability
 - Scaleability
 - Modifiability
 - Testability
- Usage Qualities
 - Dependability
 - Availability, Reliability
 - Maintainability
 - Safety
 - Security
 - Survivability
 - Performance
 - (many measures: often best handled as attribute of functions)
 - Usability
 - Interoperability
- Programme Requirements
 - Development Requirements
 - Test Requirements
 - Test Approach
 - Special to Purpose Test Equipment
 - Simulators
 - Trials & Parallel Operations
 - Costs
 - Timescales

Figure 6.3: Common kinds of non-functional system requirements.

A simplified tree of fairly typical system nonfunctional requirements is shown in Figure 6.3. A more detailed tree with hints and examples is available at <http://www.scenarioplus.org.uk>. You will certainly need to customise the tree for your project – you will need more detail in some areas, and will probably want to reorganise the tree to suit your domain.

NFRs are notably difficult to organise. The categories may overlap, and related requirements may be found in different parts of the tree. For instance,

you could have user interface constraints imposed for compatibility with other products; you could have user interface standards or guidelines; you might have human factors requirements on seating, lighting, and console dimensions; and you could also have usability targets for certain tasks.

However, despite these difficulties, the suggested headings should help you to identify missed requirements and areas needing investigation.

Using Standards to Identify Qualities

There are numerous published lists of qualities, especially for software. Some of these are in international standards. The nice thing about these, as Andy Tanenbaum¹ remarks, is that there are so many to choose from.

For example, ISO/IEC 9126 (*Software Product Quality*, 2001) lists six qualities:

- functionality (!), including security, accuracy and interoperability;
- reliability;
- usability;
- efficiency;
- maintainability;
- portability.

We consider qualities to be a different category from functions, so this part of the ISO 9126 standard feels quite confusing.

In contrast, IEEE 830 (*IEEE Recommended Practice for Software Requirements Specifications*, 1998) offers a different list, organised in another way:

- performance requirements;
- logical database requirements;
- design constraints, including standards compliance;
- software system attributes (i.e., qualities):
 - reliability;
 - availability;
 - security;
 - maintainability;
 - portability;
 - ease of use.

These lists of ‘-ilities’ (and others like them) clearly overlap, but only cover a few kinds of *software* quality. We suggest you start with the list given in Figure 6.4 for a software-only project, and customise it as necessary. Notice that

¹Tanenbaum, A.S. (2002) *Computer Networks*, 4th Edition, Upper Saddle River, NJ: Prentice Hall

- Development Qualities
 - Maintainability
 - Choice of Programming Languages/Toolkits
 - Clarity of Documentation (of Code)
 - Modularity
 - Flexibility
 - Scalability (e.g. to multiple servers)
 - Portability (e.g. to other operating systems)
 - Testability
- Usage Qualities
 - Dependability
 - Availability (of service)
 - Reliability (of the software)
 - Safety
 - Security
 - Performance
 - Speed
 - Accuracy
 - Usability
 - Operability
 - Accessibility (for the disabled)
 - User Interface Guidelines
 - Internationalisation
- Project (or Programme) Requirements
 - Costs
 - Timescales

Figure 6.4: Subset of NFRs for a typical software project.

the list distinguishes usage (product) qualities from development (process) qualities.

These software lists are explicitly *not* designed to be suitable for systems involving hardware, whether computational or not.

A lesson here is perhaps that following the crowd may not be healthy for your project. Even international standards can be full of gaps. So, let us take our own advice and suggest a software checklist to help you get started on your project.

A Software Checklist

On a typical software project (where no hardware product is being developed), the main types of nonfunctional requirement are qualities, as shown in Figure 6.4. Many of the types of constraint listed in Figure 6.3 are not relevant.

Small software projects may sometimes be concerned with very few of the qualities listed in Figure 6.4. For example, suppose you are developing an additional web application for an online retailer, using cookies (files held locally on client computers) that record items customers have looked at, to identify other items that individual customers might like to buy. The ultimate goal of this is to increase sales. You will be concerned mainly with getting the functionality to work. As for qualities in this application project,

you will perhaps only really be concerned with performance and (product) usability.

Can an NFR be Functional?

You might expect it to be easy to tell if a requirement is functional or not. In practice, the boundary is often quite fuzzy. Here are some examples of 'functional NFRs':

1. An interface connector (see Chapter 4) may seem obviously nonfunctional; it's a piece of design, and specifying it constrains the product design. But to sales and marketing, providing that interface adds a *functional feature* that helps to sell the product:
'Mini-USB connector for easy transfer of videos, photos, and music files'.
To the customer too, the interface is a key bit of functionality:
'The product lets me upload and carry my PowerPoint presentations without having to bring along my laptop ... '
2. An old underground commuter railway is too crowded, so the railway company decides to upgrade it. They have two key goals: to reduce journey time (minutes from A to B); and to increase passenger capacity (passengers per hour). They could increase capacity by lengthening the trains and platforms (very costly); by running more trains; and by running the trains faster, which also reduces journey time. All of these affect signalling, which is a set of safety-critical functions. You could say that journey time and capacity were performance targets, hence usage qualities (see Figure 6.4). But to a railwayman, these are the key *functions* of the railway, and they also imply many further functions. In fact, carrying P passengers per hour in journey time J is basically all the railway is for.
3. Large, ill-defined quality goals (at high level) often quickly break down into a mix of functions and smaller quality targets. Safety, for instance, is achieved by a combination of choosing *reliable hardware components* and adding *functions* whose explicit purpose is safety. For example, a food blender has a safety switch to cut off the power automatically when the cover is opened.
4. Even operational qualities like interoperability are hard to classify. For instance:
'The mobile phone can use any mains charger'.
The user's wish here is to be able to pick up and use safely any charger that comes to hand. This is close to being a *function* (and it is certainly a *scenario*). In practice, it will mean that all (new) phones will use the same range of charging voltages and the same physical connector. Thus, in practice, interoperability requirements appear to a project as a combination of *design constraints* and functional *interfaces*.

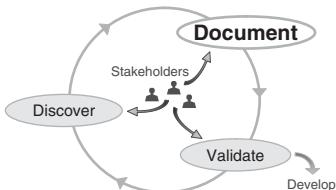
The moral of these stories is perhaps that:

- it doesn't matter what type a requirement is, function or not, just that you don't forget to include it;
- there is no point trying to force requirements language on people. When you're interviewing a stakeholder, don't start talking about 'nonfunctional goals' or 'quality requirements'. Ask plain questions about what people want to achieve, and what performance they are seeking.

Tips for Discovering Qualities and Constraints

- Customise a checklist to cover the kinds of requirement you expect on your project. For each category in your checklist, ask what requirements the project should have.
- Hold a workshop to identify the most important goals of the project. Many of these will be product qualities. Analyse these to create realistic, measurable requirements (both functions and qualities).

6.4 Documenting Qualities and Constraints



Qualities and constraints are often harder to write than functional requirements, because:

- there are many different types, so you need a lot of experience to become familiar with every possible type;
- qualities mentioned by stakeholders may be vague, high-level goals that are inherently difficult to measure and verify.

Qualities and constraints can also be easier to overlook, both for the reasons just given, and because, if you are writing use cases (see Chapter 5), functions appear as part of the flow of the scenarios, whereas qualities and constraints appear 'locally' as annexes to individual use cases, or 'globally' as an annex to the whole set of use cases.

Here are some suggestions for writing some common types of quality and constraint. The order of sections follows as closely as possible the order shown in Figures 6.3, 6.4 and 6.5; however, there are slight differences between the templates for systems and software NFRs.

6.4.1 Constraints

Design Constraints

Design constraints are very varied, and the important ones differ widely between domains. Often, such requirements are imposed to contribute towards higher-level goals (see box, ‘Towards Sustainability’ on page 144) such as safety, reduced development and running costs, and protecting the environment.

For example, in Germany, products including car parts are required by law to be recyclable. This means that it must be possible to disassemble them into components that can be recycled. In turn, that means that each component should be made of a single material such as a metal or a reusable plastic, or, if not, that a workable procedure be defined for separating the materials, e.g. using an acid to dissolve metals. Requirements of this kind concern the decommissioning/disposal phase of the whole life scenario (see Chapter 5).

A design constraint:

Beryllium shall not be used on board the spacecraft.

Comment: Beryllium is a toxic metal.

Regulations and Standards

When a constraint is already well documented in regulations or standards, the task of discovery is limited to identifying which documents (or specific parts of them) are applicable to your project.

Either way, your project need only state (with a single requirement) that ‘such-and-such’ set of regulations or standards is applicable, so that people reading your requirements know where to look.

Reused requirements will look something like this:

The product shall comply with Bluetooth v2.1 + EDR.

Avoiding Waste: Imposing as Little as Possible

Often, only a part of the regulations or standards may be applicable. You can then save yourself time and money by making clear in your requirements which sections have to be complied with, and which can safely be ignored.

COTS or Custom?

Many organisations now require their projects to try to use commercial off-the-shelf (COTS) products, such as databases, office software and personal computers, rather than specialised software and equipment, as far as possible.

- From the point of view of the organisation, this is *a way of saving money*, buying cheap commodity products instead of creating new ones.
- From the point of view of a project, this is *a design constraint* that may well not be appropriate.

The rule is hard to enforce, because it is a matter of opinion whether a COTS product that fails to meet some of the requirements, is essentially equivalent to a custom product that does meet the requirements. There is a trade-off between the cost (felt by the organisation's purchasing department) and the capability of the product (experienced by different stakeholders – operators and beneficiaries).

The use of COTS products within a larger system is *not guaranteed to save money*. The COTS product by itself may be far cheaper than the design elements it replaces. But, incorporating it may demand custom software, such as database scripts or custom hardware, to handle the interfaces created by partitioning the system into COTS and non-COTS subsystems. These may be costly:

- to create, especially if the COTS product is poorly documented (or if details of its design and interfaces are proprietary);
- to maintain, if the COTS manufacturer changes the product's interfaces from one version to the next.

Requiring the use of COTS products thus *creates new risks* for the project.

There is a corresponding risk involved in doing this in a contractual situation: if you omit an essential constraint, the contractor will ask for an additional payment for the extra work involved in complying with it. Therefore, you need people with detailed knowledge of the area to write requirements that call out specific parts of standards or regulations.

The requirements will look like this:

All electrical cables shall comply with RSE/STD/024 Part 6.

(And so on, for each part of each standard you need to impose.)

There will be a requirement of this kind for each section of the standards and regulations that your contractors must comply with.

Towards Sustainability

Design constraints can contribute to higher-level goals, such as sustainability. Sustainability is the ability of a whole system to continue indefinitely without degrading the environment. Attaining this will demand major changes to many systems.

For example, the use of glass (melted silica and other minerals) is relatively sustainable because glass can be recycled by re-melting and reshaping into new bottles and other products; the only unavoidable input is energy and, in theory, there could be no other outputs (in practice, waste materials are likely). In contrast, the use of ceramics (fired clay) is less sustainable, because once fired the materials cannot be reshaped.

As with other high-level goals on whole systems, it is not enough to say that a product 'must be sustainable'. Such goals must be broken down into realistic and measurable product constraints.

Human Factors

Human factors are important enough in some domains to be treated as a separate engineering discipline (as are safety and security). Human factors constraints include:

- (mandated) assumptions about the user/operator population, i.e. the ranges of body size, arm length, grip strength and so on among the people who will operate the product;
- constraints on console or cockpit design;
- requirements on lighting, seating, etc;
- requirements on operator selection, training and certification.

The related subject of usability is discussed in Section 6.4.3.

An example of a human factors requirement is:

The height of the display screen shall be adjustable over a range of 20 cm.

Environmental Constraints

The relationship between a product and its environment is two-way, so you should consider both:

- how much the product is allowed to affect the environment (e.g. because of the risk that other products may be nearby); and
- how much variation in the environment the product must withstand.

In the case of electromagnetic compatibility (EMC), these have names like:

- 'allowed emissions' (so as not to harm other systems); and
- 'required immunity' (from other systems' emissions).

For example, any electrical device in a car, aircraft or spacecraft must operate close to many other devices. Interference that could cause wrong (and possibly dangerous) operations must be minimised. Requiring that each device both keeps its own emissions low and has high immunity to other devices' emissions, is a two-way, 'belt and braces' approach.

Environmental constraints differ widely in importance for different kinds of product. For example, for a jet engine, reducing noise is a major goal. Modern civil aircraft engines are, in consequence, far quieter than older and military engines, especially with respect to the annoying whistling frequencies.

An example of limiting a product's effect on the environment might be:

The product shall emit no more than 5 watts of heat.

An example of withstanding variation in the environment is:

The product shall resist water ingress to a depth of five metres.

Comment: this is meant to be enough for use in rain or for accidental immersion.

Physical Constraints

Straightforward physical constraints can often be written quite plainly, because the constraining values are directly measurable and verifiable.

A simple physical constraint is:

The product, including its power supply, shall weigh no more than 600 grams.

Tips for Documenting Qualities and Constraints

- When writing qualities and constraints, consider how you will test or otherwise verify them, and make your chosen approach clear in the requirement text, verification method and acceptance criteria.
- Inspect each use case. Does it need a performance target?
- If specific use cases are critical to the business, create dependability requirements (availability, security, safety, etc) for them.
- Place qualities that apply generally (not just to certain functions or use cases) in a chapter on global requirements.

Care may be needed with hardware constraints to provide a ‘tolerance’ or range of allowable values, if the constraint is not simply imposing an upper or lower limit.

An example of a constraint with tolerances is:

Two 5 mm holes for mounting bolts shall be drilled 100 mm apart horizontally, centred horizontally and vertically on the back plate of the case, to tolerances of ±0.5 mm in both directions.

Notice how awkward it is to describe such design constraints purely in text. You can simplify the task to some extent by splitting up this constraint, first with a global rule such as:

Drilled holes shall have a tolerance of ±0.5 mm in both directions.

and then specific (local) constraints for this particular component. An alternative in this case would be to provide a technical drawing with superimposed measurements and tolerances.

6.4.2 Development (Process) Qualities

Producibility

Producibility refers to how easily a physical thing can be manufactured. Production difficulties can be caused in many ways, such as:

- by making tolerances (e.g. of variation in size) needlessly strict;
- by leaving excessively small space for access between parts;
- by having items that are unusually heavy or large (e.g. so that one worker cannot lift them unaided);
- by having items that are exceptionally delicate (e.g. fragile, or vulnerable to damage by dust or water vapour).

For example:

The casing shall be capable of being manufactured by injection moulding.

Flexibility

Flexibility means the ease with which a product can be adapted in some way. By itself, this is notoriously vague and unverifiable, so you must specify which kind of flexibility you mean, and ensure that you provide a reliable measure (see Chapter 9).

For example, upgradability can be measured:

Memory shall be upgradable to 16 Gigabytes by exchanging memory cards.

Similarly, you could specify, for instance, that a processor network could be scaled up to at least 128 processors, to yield at least 80 times the performance of a single processor. You might want to specify the required performance curve with a table or graph.

Testability

Testability means the ease with which a product can be tested, for instance in the field.

For example:

The product shall test its memory on startup.

6.4.3 Usage (Product) Qualities

Dependability

Dependability of a computing system is the ability to deliver service that can justifiably be trusted.

Avizienis, Laprie and Randell²

Dependability is a complex subject. To start with, software and hardware (computational or not) pose quite different issues for dependability. Also, many of the qualities of a system that contribute to dependability are interrelated.

Software Dependability

Software dependability essentially only means *correctness of design* (including low-level design and coding)³; once it is correct, it stays that way, at least until requirements change. To put this another way, software, unlike physical products, does not become less dependable with time; it has no gear-teeth or bearings that wear out. In contrast, time is extremely important in system and hardware dependability.

System and Hardware Dependability

Dependability is the result of combining many product qualities and system factors (such as how well trained the human operators are). Some of these are illustrated in Figure 6.5. These factors are often interdependent, and hard to separate.

²Avizienis, A., Laprie, J-C. and Randell, B. (2001) *Fundamental Concepts of Dependability*, Research Report N01145, LAAS-CNRS, April 2001

³It is possible to design systems to tolerate incorrect software, by providing a framework containing two or more alternative algorithms to achieve critically important goals, but this kind of ‘fault tolerance’ is not often attempted.

- Dependability
 - Availability
 - Reliability
 - Maintainability
 - Design choices, e.g. Redundancy, Fault Tolerance
 - Quality of Training of Human Operators
 - Safety
 - Security
 - Confidentiality
 - Integrity, e.g. Data Integrity
 - Survivability

Figure 6.5: Factors contributing to dependability.

Demonstrating Dependability

Reliability principally means a set of calculated probabilities of failure, including typically the mean time between failures (MTBF). Together with the mean time to repair (MTTR), this allows the calculation of how much of the time a product is available, and hence how many machines are needed to ensure that a satisfactory service is available.

Safety and availability can be argued in a safety case, typically supported by reliability calculations and actual test results. For example, an aircraft maker has to present a safety case to the aviation authorities to show that a newly developed aircraft is safe to fly. The safety case argues that the whole aircraft is safe because it is made of an airframe, engines and software, which work together safely, and each of these parts is itself safe.

Evidence for the safety case is provided from analyses such as the failure mode evaluation and criticality analysis (FMECA), tests on each part, and tests on the whole system.

'Type tests' may be done on a sample of components (e.g. bolts) of a given type. The results may then be assumed to hold for all components of that type.

Simulation and modelling may be used to show when failures can be expected.

Availability, Reliability

Hardware reliability means that the product has an acceptably low probability of failing in a given period of operation. Mechanical components age and then fail through wear, cracking, oxidation and other processes. Electrical and

electronic components similarly age and fail through physical processes, such as heating and radiation damage. These mechanisms reduce the reliability of products and services. For example:

The product shall have a mean time between failures (MTBF) of at least 20,000 hours of operation.

A service's availability for service depends on the reliability of all the products that it uses.

A high availability can be provided by choosing or designing very reliable components. Unfortunately, this is difficult as the overall reliability of a product or service is found by multiplying the reliabilities of its component parts, assuming they are all necessary and that they can fail independently. So if a product contains just two parts, each with reliability 0.9 (or 90%), the product only has a reliability of $0.9 \times 0.9 = 0.81$ (or 81%) for a given time.

Availability also demands that the components are reliably assembled, so that they do not shake loose, etc. This leads to physical design constraints on construction, such as:

All crimped cable terminations shall comply with Standard E6487.

Comment: to ensure they are strong and durable.

This last requirement is also an example of calling out a standard. NFRs of different types thus often belong together.

A high availability can be provided, despite unreliable components, by requiring redundancy; if one component fails, another, possibly of a different type, can continue to provide a service, possibly at a degraded level. For example:

Compass heading shall be displayed by a magnetic compass independent of system electrical power.

Maintainability

Availability can also be kept high through good maintenance. If you know that an aircraft component will probably fail after 1000 hours, you can keep the aircraft working by replacing that component more frequently, say every 500 hours. The task of managing such a programme of supply and replacement is called integrated logistics support (ILS).

For example:

The support system shall provide an ILS schedule for all assemblies.

Guest Box: Soren Lauesen on Availability

Soren Lauesen is a professor at the IT University of Copenhagen. He has worked in the IT industry for 20 years and has been a professor for 24 years. In addition to software requirements, his interests include usability, object-oriented design, quality assurance and product development.

What follows is an extract from his *Guide to Requirements* (2007), followed by the matching section of his requirements template [2].

Extract from Guide to Requirements

L2. Availability

Availability is the fraction of time where the system must be operational from the user's perspective. We have to define more precisely what it means that the system is out of operation, and how we deal with cases where some users can access the system but others cannot. If only one user cannot access the system, we would hardly call it a system breakdown.

A breakdown can have many causes and the template mentions 5. Not all of them are the supplier's responsibility. In the example it is only cause 3 (errors in software or configuration). When the supplier is responsible for the operation, also power failure, hardware breakdown, capacity problems, etc, are his responsibility.

In principle the customer can state all kinds of requirements for calculating the availability, but in practice he must accept the possibilities the supplier can offer – as long as they cover his real needs.

The introduction part suggests one way to calculate a breakdown period: A breakdown is always calculated as 20 minutes. An operational period must last at least 60 minutes. The reason is that users don't resume their interrupted tasks until around 20 minutes after the breakdown, and they cannot produce much in an operational period less than an hour.

The template also suggests a way to calculate the availability when only some of the users are affected by the breakdown.

Notice requirement 1, which says that the availability must be calculated periodically. This means that excess availability cannot be transferred from one period to the next. In column two the customer has suggested that availability is calculated as described in the introduction part. The supplier may propose his own way of calculating the availability, for instance by referring to an appendix.

Requirements 2–3 state the required availability in two different operational periods.

Extract from Template

L2. Availability

The system is out of operation when it doesn't support some of the users as usual. The cause of the breakdown may be:

1. The customer's issues, e.g. errors in the customer's equipment.
2. External errors, e.g. power failure.
3. The supplier's issues, e.g. errors in software or a wrong configuration.
4. Planned maintenance.
5. Insufficient hardware capacity.

Measuring availability

A breakdown is counted as at least 20 minutes, even if normal operation is resumed before. If the following period of normal operation is less than 60 minutes, it is considered part of the breakdown period.

Only breakdowns caused by the supplier's issues (cause 3) are included in the availability statements. If the supplier is responsible for operations too, he may also be responsible for causes 2, 4, and 5.

The **operational time** in a period is calculated as the total length of the period minus the total length of the breakdowns for which the supplier is responsible. The **availability** is calculated as the operational time divided by the total length of the period. When only some of the users experience a breakdown, the availability may be adjusted. One way is to calculate the availability for each user and take the average for all users.

Availability requirements:	Example solutions:	Code:
1. The availability must be calculated periodically. The calculation should compensate for the number of users experiencing breakdowns.	<i>The availability is stated monthly and calculated as described above.</i>	
2. <i>In the period from 8.00 to 18.00 on weekdays, the system must have high availability.</i>	<i>In these periods the total availability is at least ___ %. (The customer expects 99%)</i>	
3. <i>In other periods the availability may be lower.</i>	<i>In these periods the total availability is at least ___ %. (The customer expects 95%)</i>	

Safety

Safety means that the operation of a system (including people) carries an acceptably low probability of death, injury or damage to property.

Safety depends on many factors that are often specific to the type of product. Fortunately, safety standards exist for many types of product. For example:

The product shall comply with AS/NZS 60335.2.9:2002 for portable cooking devices.

System safety also depends on correct usage by properly trained and skilled operators (e.g. an aircraft's pilots know what they can demand from engines and airframe). For example:

All aircrew shall be trained to CAA standards for their roles.

Security

Security consists principally of two things:

- **confidentiality** – that unauthorised eyes do not get to see information that they should not;
- **integrity** – that systems, and especially their data, are not damaged, especially by unauthorised intruders, but also by unintended actions by authorised users and, indeed, incorrect design. If damage by physical causes is included here, then integrity includes at least some aspects of survivability (see the next section).

Security requirements are difficult to write, because the high-level goals are unverifiable, while lower-level options tend to be specific design choices, which may or may not achieve the desired security. Lauesen (2007) [2] suggests instead listing the specific threats that are to be countered.

An unverifiable security goal is:

Users' data is to be kept private.

(No good as a requirement.)

Examples (in the style of Lauesen) include:

Users' data is protected against the following threats:

1. *access by other users*
2. *access by non-users*
3. *access from the Internet*
4. *hard disk crash*
5. *fire.*

Survivability

Survivability is the ability of a system to continue to function under specific adverse circumstances.

If a product (e.g. a military vehicle, a building, a power station) is attacked successfully, whether electronically, by a security breach or by direct military action, it is obviously no longer available to provide a service. The same applies if service is interrupted by fire, lightning or other accidental causes.

Survivability is related to both availability and safety. For example:

The reactor casing shall survive the impact of an aircraft weighing up to 600 tonnes travelling at a speed of up to 1000 km/hour.

Survivability

Survivability used to be a purely military requirement, but given how critical information systems are to businesses today, it has become a wider concern.

For example, a bank may demand spare (redundant) copies of all its data, as well as alternative (standby) processing facilities, so it can survive the loss of any building to fire or terrorist attack with only a short period of disruption. Business planners talk of 'disaster recovery' and 'business continuity' for such functions.

The Internet is an excellent example of a communications system able to survive the loss of any single node: messages are automatically routed through whichever routing nodes are available.

Performance

Performance targets present some serious difficulties:

- performance is highly variable between runs of the same software;
- performance is equally variable between different manufactured products of the same type;
- variation is often caused by factors outside anybody's control.

For example, a system that uses the Internet for data traffic cannot guarantee how long a particular packet will take to arrive, nor whether it will arrive at all. Similarly, tasks arriving from outside, or tasks involving a human operator, cannot be estimated exactly.

Even inside a system, such as processing data requests, the time taken is not well-behaved – performance depends on how many requests arrive at once.

Therefore, statistical measures ('95% of ... ') are needed.

A Risky Performance Requirement

A software house was never paid for a major contract, containing thousands of requirements, because of one incautiously worded performance requirement. The contract said that all searches must complete within 2.5 seconds. A trial search in Singapore for the common Chinese name 'Cheng' took nearly 3 seconds, and this proved impossible to reduce. The customer got the software for free.

It may be possible to set up a 'best efforts' contract in which the software developer is paid according to how well the software performs in such cases. However, this is risky; the developer might choose not to spend a lot of effort on difficult but important cases, so the client might not get a satisfactory product.

A statistical performance target is:

95% of simple name searches on the database shall complete within 2.5 seconds.

The relationship of performance to functions is discussed in Chapter 15.

Measures of Performance: Possible Trade-offs

Performance can be measured in many ways; it has many subtypes. For example, you could have a trade-off between speed and accuracy: a machine could do a task slowly but accurately, or faster but less accurately. In that case, you could specify performance as a realistic combination of (say) a moderate speed and a moderate accuracy, using two requirements.

Other general types of performance include: network capacity or communications bandwidth; number of simultaneous transactions that a server can handle; and time to complete a transaction (including queuing).

There are any number of specialised measures of performance, depending on what you are trying to achieve. For instance, the performance of a pump may be measured in litres per second.

Usability

Usability has many of the same problems as performance. All people are different, and what is easy for some may prove difficult for others.

Be careful about breaking down usability goals too far. There is a danger of diving into premature user interface design (and writing it into the requirements) without testing it for usability. You should rarely, if ever, design the user interface while discovering requirements. As a general guideline, stay at the level of tasks performed by human operators.

The best advice is to keep usability requirements simple, and to think carefully how you will test them. Statistical targets demand extensive, costly and time-consuming testing. A requirement of 95% means that 1 failure in 20 is acceptable. That, in turn, means you need to do more than 20 tests, and perhaps 100 or more, to get an acceptably accurate result.

Usability Panels versus Usability Testing with Ordinary Users

A usability requirement:

The Usability Panel shall be able to complete task ABC in three minutes.

Definition: Usability panel = the team of judges assigned to the task of assessing the usability of system features.

A statistical usability target:

95% of untrained users shall be able to complete task XYZ in three minutes.

Note the difference between these approaches:

- A usability panel may contain people with expertise in human factors or the specific domain (air traffic control, automotive, etc), but even they may not predict the usability problems that ordinary users encounter.
- Usability testing means observing people actually using a product under properly controlled conditions, so you can see exactly where they get confused or stuck. For example, a usability laboratory may film a computer screen at the same time as the operator's face so you can see what was happening. Some labs monitor the direction of the operator's gaze and superimpose that on the image of the screen.

Usability testing is the only reliable way to measure a product's actual usability.

Can a Style Guide Solve Your Usability Problems?

It is amazingly difficult to check that a guideline is followed.

Soren Lauesen [2]

You may be able to avoid some usability issues by asking for compliance with user interface guidelines.

A user interface guidelines compliance requirement:

The user interface shall comply with the Microsoft Style Guide for Windows version NNN.

Such compliance does not guarantee that your product will be usable. Style guides are complicated; they provide many options, and some will be much less good than others for certain tasks. And, as Lauesen warns, compliance requirements are costly to verify.

Internationalisation

An internationalisation requirement:

The user interface shall be able to operate entirely in each of the languages listed in table L-1 without changing the software.

The acceptance criterion for this requirement is most naturally a test that the whole user interface can be transformed into a fresh language – say, Lithuanian, Thai or Japanese – without any further programming. However, the first languages chosen will probably be those with the largest markets, rather than those that are hardest to handle on the user interface. So, testing may instead focus on samples of the user interface to show that, when the time comes, all the languages will be handled correctly.

Internationalisation: A Usability Challenge

Usability may demand careful design across a whole system. For instance, internationalisation of software means replacing every piece of text – labels, menus, error messages, etc – with text in another language. Switching languages therefore means being able to retrieve every message using some indexing data, such as a message number and a language identifier. It also means ensuring that there is sufficient space for the most verbose language (German often takes more room than English, for example). As if that wasn't enough, some languages use letters not found in standard ASCII, so space has to be allowed for larger character sets and typefaces; and some languages run wholly or partly from right to left, too. So, adding just one requirement may change the whole style and structure of the code.

Interoperability

Interoperability, the ability of different products to work seamlessly together, is important in military and transportation systems. For example, each of land, sea and air forces must have radios that can communicate with the other forces; all the different types of train on a railway network must be able to operate correctly with all the signalling equipment. But interoperability is often given

small importance in commercial products. Companies may be more concerned to differentiate their products from their rivals than to share parts with them. So, for example, cars made by different manufacturers share very few parts beyond simple light bulbs, tyres and bolts, and even these occur in many types, despite years of attempts at standardisation.

For example:

The configuration data file for all products in the company's product line shall be structured as defined in Company Standard D12.

Comment: This is for use by the common product configuration utility.

Programme Requirements

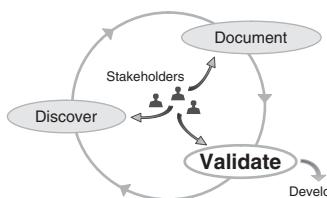
Programme requirements include any constraints on the project, notably including costs and timescales, but also including anything needed for development, test, and bringing the system under design into service. So, for instance, any special-to-purpose test equipment needed can be called out here.

For example:

A network simulator shall be provided by the start of project phase 2.

Costs and timescales are subject to such frequent change that people are understandably reluctant to put them into requirements documents; and there is no need to do so. They can be documented in project plans, contracts etc, to which the requirements can be attached.

6.5 Validating Qualities and Constraints



Getting the qualities and constraints right on your project need no longer be a 'black art'. You can conduct several simple checks on the completeness and correctness of your requirements. These checks are powerful because they are independent of each other.

1. **Verifiability:** Are all the requirements verifiable as written? Can you see from the acceptance criteria how you will test or otherwise prove they have been met? Are the tests realistic and affordable?

2. **Goal satisfaction:** Do the requirements together achieve the goals of the project? If all the detailed sub-goals were implemented as defined, would the top-level goals in fact be satisfied? It is easy to write many small requirements, but do they add up to what stakeholders actually want?
3. **Checklist completeness:** Use the requirements checklist that you have customised for your project. Are there requirements in every class? If not, are you sure you don't care about the missing classes? Are the requirements in each section enough to meet the goal of the section? (Note that the name of each checklist class implies a possible type of goal for your project, e.g. 'Availability' implies a goal like 'Service is available 99.9% of the time'. To meet this goal, individual parts of your product may be required to have higher availabilities than that.)
4. **Targets predicted to be met:** Do your simulations and models predict that your reliability, performance and other targets will be met?
5. **Qualities and constraints specific to a use case:** Does each use case have the local qualities that it needs (e.g. performance, security, usability)? If you do have local qualities, have you avoided overlaps with global qualities? Can any global qualities be dropped in favour of (possibly new) local qualities?

Tips for Validating Qualities and Constraints

- Use your customised checklist. Ask yourself whether the requirements in each category meet the actual needs of the project.
- Read each requirement. Make sure you can see how it will be measured and verified in practice.
- For the key targets like performance, reliability, safety and mean time to repair, go over the calculations and reasoning. Check that the figures make sense, and can be attained in practice.
- Check the global qualities against the list of use cases. If any of the NFRs can be limited in scope to just one use case or one group of use cases, move them there as this will save money and effort.
- Check that the requirements satisfy the project's (higher level) goals.

6.5.1 Things To Check Qualities and Constraints Against

- Nonfunctional goals (Chapter 3)
- Standards, regulations, and templates/checklists (Chapter 6)

- Qualities and constraints used on earlier projects (Chapter 13)
- Qualities and constraints mentioned in interviews (Chapter 11) and workshops (Chapter 12)

6.6 The Bare Minimum of Qualities and Constraints

- Find out the essential qualities and constraints affecting your product – the ones that make up the buy/don't buy decision for your key stakeholders.

6.7 Next Steps

- Analyse your quality requirements to discover goals (Chapter 3), most likely functions, that will enable your system to attain those qualities, and scenarios (Chapter 5) to implement those goals. Note that this will often demand a knowledge of the chosen design, discovered through optioneering (Chapter 14).
- Document the rationale (Chapter 7) for any contentious, costly or mission-critical requirements, and their measurements (Chapter 9). Safety and reliability targets are good candidates for this treatment, as they will drive many aspects of the project (and therefore its cost, risk and timescale).
- Add any terms you need to define to the project dictionary (Chapter 8).
- Ensure your requirements are measurable (Chapter 9).
- Prioritise your requirements (Chapter 10).

6.8 Exercises

Choose a high-level quality target for the outdoor leisure activity multi-function device from Section 6.3.1, or for a product that interests you:

1. Analyse goals that contribute to achieving your chosen target. Identify carefully their types, noting that some will be functions.
2. Write (realistic, measurable) acceptance criteria for the nonfunctional goals that you identified in question 1.

6.9 Further Reading

1. Lauesen, S. (2002) *Software Requirements, Styles and Techniques*, Harlow: Addison-Wesley.

In Lauesen's sensible and practical textbook is one of the best discussions of qualities and constraints. It is focused on software, so it does not cover system qualities and constraints. The 'styles' in the title refers to sentence patterns appropriate to formulating different types of quality, such as usability and security.

2. Lauesen, S. (2007) *Guide to Requirements SL-07: Template with Examples*, Copenhagen: Lauesen Publishing.

The 'guest box' extract in this chapter is taken from this more recent publication.

You can download slides, a course plan (curriculum) and templates from Lauesen's helpful website at <http://www.itu.dk/~slauesen/SorenReqs.html>

CHAPTER SEVEN

Rationale and Assumptions

Of all the things you can do to improve your requirement management process, none can be done cheaper or faster or have more impact than capturing the rationale for each requirement.

Ivy F. Hooks and Kristin A. Farry

Requirement Elements	Rationale and Assumptions
Discovery Contexts	
Introduction	
From Individuals	
From Groups	
From Things	
Trade-Offs	
Putting it all Together	
Priorities	
Measurements	
Definitions	
Qualities and Constraints	
Scenarios	
Context, Interfaces, Scope	
Goals	
Stakeholders	

Answering the questions:

- Why (did you) choose to do it this way?
- When would you have to rethink?
- Where did this come from?
- Why is this needed?
 - ... to help reach good decisions on requirements and design approaches
 - ... to protect important requirements from being dropped
 - ... to resolve disagreements
 - ... to permit intelligent reuse
 - ... to make hidden assumptions explicit
 - ... to trigger a prompt rethink if a vital assumption breaks.

7.1 Summary

Explaining the rationale for at least the most contentious requirements and design decisions on a project saves time, confusion, and money.

Decisions on requirements and design are often based on hidden assumptions. Making assumptions explicit, and connecting them to an argued rationale, enables decisions to be revisited without starting all over again.

A rationale can be explained by stating reasons as text, by listing or modelling assumptions, or by tracing decisions back to other items, such as goals.

Traceability means enabling the people on a project to trace forwards or backwards between items and their causes. For example, a requirement can cause a project to create both design elements and tests. A set of design elements can implement a requirement. A set of tests can verify a requirement. The traces back to the requirement act as the rationale for both design elements and tests. The relevant requirements justify the choices made. Traces help people to find relevant project information quickly.

An understanding of rationale enables projects to prioritise accurately. It protects essential requirements from being deleted. The rationale explains the decisions made on the project. The rationale also helps people joining a project to understand what it is about.

7.2 The Value of Rationale

Nothing can be lasting when reason does not rule.

Quintus Curtius Rufus

Ironically, a rationale may need rather more justification of its own (a rationale for having a rationale) than the other elements described here in Part I. This

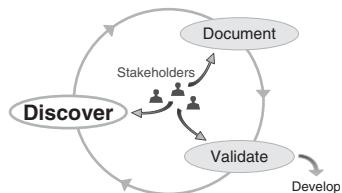
is perhaps because the rationale is, necessarily, secondary to whatever is to be justified, whether that is a requirement, a design decision or a priority. In a busy project, getting the primary things defined is work enough.

However, there are good reasons to value rationale within a project and within a business. Rationale helps you:

- to come to a correct decision:
 - on requirements or design (see Chapters 10 and 14);
- to show a decision was correct (founded on reasonable assumptions):
 - e.g. when a project is audited;
- to prevent and, if need be, to resolve conflicts;
- to allow a project's thinking and decision-making to be revisited:
 - e.g. when a 'key, loadbearing' assumption breaks and the project must be replanned;
- to reuse individual requirement elements:
 - by explaining the reasons for and against elements such as use cases or features. People do not like to reuse elements that they can't see the purpose of.

These are powerful advantages. Rationale is worthwhile, even within a single project (the first four of the above reasons). A product line involves different projects creating products with overlapping sets of features. In a business operating a product line such as this, a properly documented rationale brings additional savings 4.

7.3 Discovering Rationale and Assumptions



The rationale for a project decision, such as to include a requirement, is rarely a matter of certainty. Decisions are based on imperfect knowledge of the world and the market, and on reasoning from more or less explicitly stated assumptions about the current and future situation. In other words:

Explicit Assumptions + Clear Reasoning → Justifiable Decisions

You can start to explore the rationale of a requirement or of a whole project with a set of simple techniques for discovering hidden assumptions. In *Assumption-Based Planning* (2002) [1], James Dewar collected several of these techniques:

- asking why;
- looking for the word ‘will’ in vision statements, plans, etc;
- rationalising a set of requirements (thinking out what the reasons must have been for writing them);
- inverting risks.

Let us look at each of these techniques in turn.

7.3.1 Asking Why

The most direct way to find out why something exists is simply to ask why people want it.

Suppose there’s a requirement:

The data are to be held in an Oracle database version 9 or above.

You could accept this as a sensible choice – Oracle is certainly a fine database. Or you could try to find out why the stakeholders need this. Every choice comes at a price, not only in licensing but in administration, training, and so on. The answer might be:

‘Oh, we assumed that would be the most portable solution.’

In that case, the real requirement may be for portability; but to what? Perhaps it is to two or three specific platforms. If so, the requirement is for the data (and the software?) to be held on those platforms, and there may be several choices of database available:

The data shall be held in a relational database.

The database shall be able to run on Linux.

The database shall be able to run on Windows Vista.

Possible solutions:

- Oracle version xx
- ...

From Features with Hidden Assumptions to Stakeholder Requirements and Explicit Rationale

Asking ‘why?’ can be used in this way to sharpen up stakeholder requirements by removing premature design decisions.

Where the original requirement asked for a specific feature, implementation or commercial product, the new requirement typically asks for a function or quality of the product to be built. Where the assumption embodied in the original requirement was unstated, the new requirement states it explicitly, and creates some freedom to rethink the assumed solution.

Asking ‘why?’ is not always the right thing to do.

- If people have taken trouble to define a specification, they may find it rude or threatening to be asked why they have written each statement. Asking the question in a different way, such as: ‘When do you need this?’ or ‘Talk me through how you do this,’ often leads to an explanation.
- Asking ‘why?’ essentially searches for causes at a higher level. Where the requirements are at the level of a system, the higher level may be the mission statement. This may or may not be worth looking for. If the answer to ‘why?’ is ‘because that’s what the customer wants’, then that part of the rationale is complete.

Other ways of discovering rationale may be more appropriate in such circumstances.

James Dewar’s (2002) excellent book [1] is not about requirements or product development at all, but it contains such a well thought out set of cognitive ‘tools’ for finding out the reasons why people have done things, that it is an invaluable source of ideas, including the following, which work well on development projects.

7.3.2 Looking for the word ‘will’ in vision statements, plans, etc

One very general technique is to look through memos, vision statements, plans, technical notes and similar documents for giveaway words like ‘will’. For example:

The system will be running alongside the existing customer management system

...

The unstated or ‘tacit’ implication behind the ‘will’ is:

‘We assume that the system will be running alongside ... ’

Now this is very interesting. It at once leads to some good questions:

- What depends on this assumption?
- Why have people made this assumption?

You can see that assumptions can underlie other assumptions, forming chains of reasoning. In other words:

Explicit Assumptions + Connections Between Them → Rationale

Is rationale always composed of chains of assumptions? Not necessarily (see Section 7.4, Documenting Rationale).

Assumptions are important, because:

- some of the assumed reasons may be missed requirements;
- some may be things that are wrong;
- some may be true today, but may be out of the control of the project, so if you depend on them and they break, the requirements will have to be changed.

7.3.3 Rationalising a Set of Requirements

Sometimes you will be asked to review a set of requirements because people feel there is something wrong, but can't say quite what the matter is.

For example, a project may take over an old specification, with the intention of updating it. In this situation, the requirements are hard to evaluate because you don't know what is assumed, or what the reasons are for the unexplained statements of need.

You can move the project towards a manageable set of requirements by deliberately inventing the reasons that could have led to the requirements as stated. You can then share these possible reasons with stakeholders, and agree which ones actually matter on the project. These will lead you towards the project's real goals (see Chapter 3). You can then see which requirements contribute to those goals, and weed out those that don't. You can also see which goals are not fully implemented by the requirements, and fill in the gaps. That way, your review will be far deeper and more effective than usual; it will discover and fix the real weaknesses in the requirements.

Example: Rationalising a Multifunction Device's Specifications

Suppose a specification states the requirements for a device, without explanation (Figure 7.1).

Even with this brief information, we can see that the device is a multifunction product intended for outside use. The five-minute reading light is interesting; perhaps it is for emergency use, along with the GPS (Global Positioning System) navigation. If so, the product needs to be very reliable. It sounds as

Requirements (not-yet-justified features)

- The device shall be portable.
- The device shall provide GPS location accurate to the nearest 100m.
- The device shall provide a compass heading accurate to 0.5 degrees.
- The device shall be water-resistant.
- The device shall receive FM radio.
- The device shall provide light sufficient to read by for at least five minutes.

Figure 7.1: (Example) Requirements listed without justification.**Inferred Assumptions**

- Many people enjoy outdoor leisure activities.
- There is a market for multi-functional devices for outdoor leisure use.

Inferred Goals

- The company wants to sell multi-functional devices for outdoor leisure use.
- The device will be:
 - carried on the person.
 - hand-held.
 - relied on in emergency for a range of functions.
- The device may be used:
 - in all weathers.
 - high in the mountains.
 - in a canoe or small boat.
 - for some days away from mains electrical power.

Figure 7.2: Inferred assumptions and goals to be checked with stakeholders.

if this product is aimed at the leisure market, perhaps for mountain-walking, camping, fishing, cycling and so on. We are making educated guesses about the unstated but assumed goals for the product.

We can't be sure about these guesses, but if they are right we will be well on the way to understanding the company's goals for the product. So we rationalise, as in Figure 7.2.

Clearly, this list could be extended. Each of these guesses about what people's goals 'really are' could be wrong, but they are all plausible and, crucially, they can be checked with stakeholders (none of them are in technical language). If they are shown to be right, they will quickly lead to new (missed) product requirements, such as those in Figure 7.3.

All of these need further work. In fact, the rationalisation exercise is useful partly because it shows how much more work is needed to bring the requirements to a usable state of completeness.

Possible New Requirements

- The device shall weigh no more than (TBD 800) grams.
- The device shall be operable between (TBD -50) and (TBD +60) degrees Celsius.
- The device shall be waterproof to (TBD 3) atmospheres pressure.
- The device shall survive a drop of (TBD 1) metre onto a hard surface.
- The device shall be rechargeable in the field in (TBD 30) minutes.
- The device shall have an MTBF of (TBD 10,000) hours.

Figure 7.3: Missed requirements from the inferred goals.

All the ‘TBD’ (to be decided) quantities (see Chapter 9) in Figure 7.3 need to be evaluated. That can’t be done purely by getting stakeholders to say what they would like (even if they were able to come up with such numbers); it is a trade-off between what is wanted and what can be achieved using available technology. Trade-offs are discussed further in Chapter 14.

The requirement to be rechargeable in the field is interesting. In a way, it is complete as a pure goal, but as a system requirement it needs to be quantified. For example, it could be:

... sufficient to operate for 30 minutes after five minutes' winding

but that begs the question of whether the recharging is to be done by winding a handle, by a small panel of solar cells, or whatever. Since ‘in emergency’ can include ‘by night’, winding would be better than relying on sunlight. This suggests it would be wise to make explicit that recharging should not require sunlight, and should be quick.

It is plainly impossible to state all possible assumptions explicitly; you’d go on for ever. It is also impossible to keep the requirements free of all design decisions, like whether there should be a winding handle or a solar panel. The ultimate rationale for a project – its mission statement – should be far back from design. But the requirements themselves can often only really be stated in design terms. Your aim should be to avoid making unnecessary or unjustified design choices. Finding the rationale for the requirements will go a long way to helping you do this.

7.3.4 Inverting Risks

If, in a technical note somewhere, you find a statement like:

Provided the device is warm enough for the processor to operate correctly, ...

then you can at once see that someone is worried by unstated but conflicting assumptions about the processor’s temperature limits and the actual environment (Figure 7.4).

Assumptions

- CMOS processors cannot operate below -20 degrees Celsius.
- Temperatures in the mountains are as low as -50 degrees Celsius.

Figure 7.4: Assumptions discovered by inverting risks.

The risk that the processor's limit will be broken could be handled by writing a warning in the user manual:

*Warning: in extreme cold, take care to keep the device warm, e.g.
by carrying it inside your clothing. It may not operate correctly below ...*

Of course, users do not spend much time reading such things (ironically referred to by developers as 'document engineering'), and warnings do not help users in emergencies.

A better answer is to add a requirement to handle the risk:

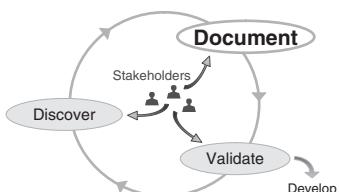
The device shall be operable between -50 and +60 degrees Celsius.

This is effectively a business requirement, derived from the expected conditions of use of the product on icy mountains, in hot deserts, etc.

Together with the now explicitly stated assumptions about the processor's lower temperature limit and mountain temperatures, this requirement might lead to design options such as insulation and heating to keep the processor at a safe working temperature. These in turn could become requirements. The process of discovering requirements by evaluating design options is described further in Chapter 14.

Notice that, in this example, both the assumptions and the requirement are necessary.

7.4 Documenting Rationale



The way you choose to document rationale depends on the nature of your project, the tools you are using, and what you want to do with the rationale.

Proven options include:

- justification text (e.g. a database field) for each requirement;
- lists of assumptions, risks, issues and decisions;
- a project dictionary (list of terms and their definitions) to explain specialised meanings, data structures and roles/responsibilities;
- rationale models, for the most contentious requirements;
- tracing back to goals, risks, assumptions, etc that justify the requirements (i.e. without any explicit ‘rationale’ text);
- specialised argumentation notations, e.g. GSN and CAE, to make the safety case for a product that needs a safety certificate from a regulator. (GSN is briefly discussed in this chapter, as an example. See also Appendix C, Tools.)

On a large project, you might use several of these (Figure 7.5) in different places. On a small project, you might just list your assumptions and risks, and define the terms in a project dictionary.

Note that most of these options rely on traceability to connect requirements to rationale (in whatever form). We'll look at each of these approaches in turn.

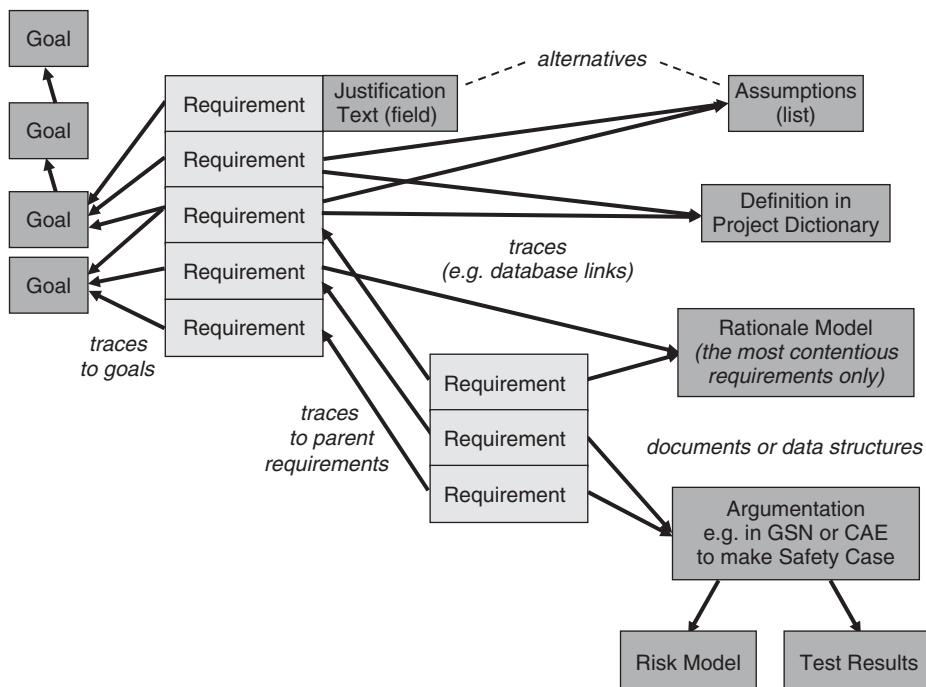


Figure 7.5: Possible ways to document project rationale.

7.4.1 Justification Text Field

A simple way to record rationale is to provide a ‘justification’ text field as part of each requirement (Figure 7.6). This means that a requirement becomes a set of related facts (what database people call a ‘record’) like a row of a table, rather than just a string of text.

This does not absolutely presuppose that you are using a requirements database tool, though such a tool makes it simple to show or hide the justification field as desired. You could use a Word table or an Excel spreadsheet, for example, to handle columns for justification, priority, etc.

Unfortunately, Word makes it awkward to work with many columns, as these cannot conveniently be hidden or shown as desired; in the example shown here, just the six columns shown are already close to the limit for standard ‘portrait’ page layout.

Similarly, Excel makes it awkward to work with section headings and diagrams, and large documents become difficult to handle and print. Sharing also becomes an issue; only one person can edit a document at a time. A practical limit with Excel is about 750–1000 requirements. Hence, writing justifications as text fields – requirement attributes – is only a sufficient means of recording rationale on small projects.

A bigger problem with the use of a justification attribute is that the same rationale may justify many different requirements. Worse, the same assumptions may, in different combinations, make up the rationales for different requirements. For example, the justification for waterproofing could be: ‘Active outdoor usage. Device may get wet in rain, rivers, etc.’, which shares the assumption ‘Active outdoor usage’ with the requirement in Figure 7.6. In other words, the rationales overlap.

You can repeat a rationale text by copying and pasting, but this makes the document repetitive and dull to read, and dangerous to edit; there is the constant risk that you will update some but not all of the copies, leading to inconsistency and error. Traceability between separate lists of requirements and of assumptions is more trouble to set up, but is safer.

ID	Description	Justification	Owner	Priority	Status
S-67	The device shall survive a drop of 1 metre onto a hard surface.	Active outdoor usage. Knocks are likely.	Product Management	Very desirable	Draft

Rationale as a piece of text within a requirement record

This Rationale is made up of two Assumptions

Figure 7.6: Rationale as a ‘justification’ text field in a requirement record.

If your project runs up against such limitations, you should consider using a requirements database tool to automate your traceability.

Why Duplicates are Dangerous

If you are used to managing information using ordinary documents and spreadsheets, you will know that people tend to look at each document by itself. They may print it off and read it on the train, for instance. You therefore feel you have to put in everything anyone might need to know from other parts of your project – the basic purpose, the timescale and budget, the glossary, the latest version of the project's table of risks, whatever. This means that each document contains a lot of information that duplicates things in other documents.

To be more precise, it *approximately* duplicates things: each time a new document is created, a few changes are made. Soon, there are many slight variants of each requirement, assumption and every other type of item, all inconsistent with each other. Going back through all the old documents to remove the inconsistencies is a nightmare, but leaving a set of requirements inconsistent is a recipe for confusion and delay.

The same happens if you have a long list of requirements and there are many duplicate reasons in a 'justification' field. Making a change means doing a global 'search and replace' in the list. People will often forget and make a local change, creating inconsistencies. In short, duplicates are dangerous.

7.4.2 Lists of Assumptions, Risks, Issues and Decisions

A traditional and still effective way of making project rationale explicit is to document different requirement elements, such as assumptions, as lists. Lists may be a sufficient means of recording rationale for small and short-lived projects.

Some of these go together in a natural way:

- An **assumption** is something you cannot control, but take to be true (for the time being).
- A **risk** is something that would be a problem for the project, for example if an assumption failed. Hence, people often list assumptions and risks together.
- An **issue** is a question or problem that needs to be resolved to make the requirements, design, test approach (etc) definite. An 'issues' list is a natural way for a project to identify which items need a rationale.

- A **decision** is a resolution of an issue. Hence, people often list issues in one column and decisions in another alongside it.

Paired lists have a serious limitation, however; items often do not relate to just one other item. For example, a decision sometimes resolves multiple issues; a risk may need to be mitigated by several requirements. That means that, to avoid duplication, you should keep the lists separate and trace the related items to each other, as with requirements.

7.4.3 Traceability to Goals, Assumptions, etc

The traditional way to show the reasons for a requirement is to trace it back to the goals, assumptions or customer statements that brought it into being. That way, you have a simple, clean list of goals, another list of requirements, and so on. An item like a requirement can have as many traces as you need to give it, with no duplication.

Traceability is appropriate (and not only for documenting rationale) when the number of requirements, and the number of people on your project, makes it impossible for everyone to keep the reasons for including each requirement in their head.

For example, if your customers say:

Cust-126: The operator shall be able to use the device in the dark.

this goal statement, once accepted, creates (with human help) the system requirements:

Sys-240: The controls shall be illuminated.

Sys-241: The display shall be illuminated.

The customer goal also justifies both these requirements, as long as you connect the requirements to the goal. The answer to the question: 'Why do you have this requirement?' is transformed to: 'Where did this requirement come from?' (Figure 7.7).

Traceability is more difficult to manage than simple text and pictures, because each statement has to be individually indexed, and related statements have to be linked together directionally: 'this one derives from that one'.

ID	Description	Traces back to:
Sys-240	The controls shall be illuminated.	Cust-126
Sys-241	The display shall be illuminated.	Cust-26

Number typed in
(wrongly) "by hand"

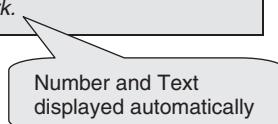
Figure 7.7: Traces connect requirements to customer goals.

It is not very pleasant handling traceability ‘by hand’:

- With large numbers of traces, it is easy to make clerical errors (e.g. ‘Cust-26’ instead of ‘Cust-126’), which may be hard to detect.
- More importantly, when someone changes a requirement in one document, they are likely to forget to update (or fail to find) all the other documents that refer to that requirement, causing inconsistency.

Requirements database tools get over this problem by showing you the text of linked items as if they were together, even though they are in separate parts of the database (Figure 7.8).

ID	Description	Traces back to:
Sys-241	The display shall be illuminated.	Cust-126: <i>The operator shall be able to use the device in the dark.</i>



Number and Text
displayed automatically

Figure 7.8: Rationale by displaying text of traced items beside a requirement.

This is much easier to use. To create a trace, you mark the items that belong together (for example, some tools allow you to drag-and-drop a requirement to a customer statement to link them together). To check a trace, you simply read the texts side by side, and ensure that the traced-to statement actually does justify the requirement. You can also usually select just the requirements that deal with a particular subject (such as the display), so that you don’t have to look at the whole specification at once. The price for this convenience is the expense and effort involved in setting up, learning and using a suitable tool.

Traceability to Parent Requirements

Tracing to parent requirements is probably the commonest case in requirements *management*, for example when subcontractors demonstrate their subsystems’ compliance with a main contractor’s system requirements. That takes place when requirements *discovery* is essentially over, and mainly consists of checking for gaps in the requirements through analysis and modelling.

Handling requirements traceability is the primary function of requirements management tools (see Appendix C).

If you’ve traced your requirements to goals or to parent requirements, then those traces form a large part of the justification for the requirements, and you will not need to create explicit rationales for the simpler requirements.

- You may want to write a short statement for every requirement, just to keep things clear and tidy.
- You may just document the rationale (e.g. with a rationale model), for the tricky cases where a decision had to be thought about in detail.

Traceability is a very powerful mechanism, as you can use it to relate any number of items of any number of kinds together. Most of these can serve as rationale:

- technical terms with their definitions in the project dictionary;
- requirements with the goals that justify them;
- tests with requirements;
- design elements with requirements;
- issues with decisions;
- assumptions with risks;
- requirements with constraints.

Satisfaction Argument

A traceability tool permits a very strict form of rationale analysis: each decomposition of a higher-level requirement into a group of lower-level requirements is explained in words as well as traced (Figure 7.9). The text, known as a satisfaction argument [7], specifically explains why the traceability structure works.

In fact it works both ways: forwards, to show how lower-level requirements and design elements are sufficient to meet higher-level requirements and goals; and looking backwards, to show why the lower-level elements are necessary.

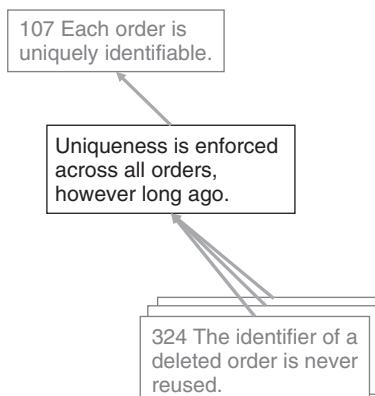


Figure 7.9: Traceability via satisfaction argument.

The advantages of this are:

- The rationale is stated explicitly.
- Rationale is tied directly into the structure of the requirements database.
- In a safety-related domain, such as transportation or nuclear power generation, this very careful arrangement helps to ensure that requirements are complete and correct, i.e. fully justified.

The disadvantages are equally clear:

- Constructing a satisfaction argument to explain every satisfaction trace is a great deal of work.
- The work is skilled; to write helpful satisfaction arguments, the analyst must understand the arguments, which lie in the particular domain, not in the world of software. The analyst must also be motivated to explain the arguments well, or else the texts will just repeat the requirements without explaining anything.
- Since the satisfaction argument lies in between the two sets of requirements, tracing between them demands two jumps, not one. Reading such a specification structure is harder than reading an ordinary requirements document, demanding deeper understanding.
- Despite the effort involved, the result is still text. This may be less clear than modelling rationale graphically (see Section 7.4.4, Rationale Models).

Choosing to model rationale with satisfaction arguments is, for all these reasons, a serious step, and one which will not suit all projects.

Over-Complex Information Models

There is a danger that clever and enthusiastic engineers may develop over-complicated structures of information (information models), with many different documents or tables elaborately traced to each other.

Over-complex structures can make requirements harder to manage:

- mistakes and missed requirements may become hard to discover in the complexity;
- basic tasks like printing a specification can become difficult;
- you can waste a lot of time polishing and refining the structure, rather than getting on with discovering the requirements.

The rule is to develop the simplest possible information model to deliver what your project actually needs.

Plato 0, Toulmin 1

Classical reasoning, as pioneered by ancient Greek philosophers like Plato, relied on logical (mathematically certain) deduction. The flow of reasoning moved from stated premisses (rules and facts) to uncontested conclusions (as long as you agreed with the premisses). A typical pattern of classical reasoning is the *syllogism*, like this:

(Rule) 'Substances are solid below their freezing points.'

(Fact) 'Water's freezing point is 0° C.'

(Conclusion) 'Water is solid below 0° C.'

This sort of reasoning is quite limited in practical use (e.g. to justify requirements) because uncertain assumptions are much more common than certain premisses, and conclusions rarely follow certainly from assumptions, either. Classical logic replies: 'In that case, you can't conclude anything certain at all', and gives up.

Happily, there is a better answer. Stephen Toulmin, in his readable book *The Uses of Argument* (1958) [2], says we can perfectly well reason from uncertain premisses. We just have to accept that our conclusions will 'most probably' hold, just as courts of law arrive at judgements 'on the balance of probability' or 'beyond reasonable doubt'. Certainty is not available in law or in business. Toulmin calls this 'substantial' reasoning as it applies in a practical way to things rather than pure ideas.

The rationale models in this chapter use a simplified form of Toulmin reasoning: all the components of an argument are treated as assumptions, with connections between them. We begin with assumptions that support the conclusion (Toulmin calls these the *warrant* for believing the conclusion) and, if necessary, also document assumptions that tend to contradict the conclusion (which Toulmin calls the *rebuttal*).

In practice, the conclusions that projects are interested in are those that translate to practical goals: an assumption is something you take to be (very probably) true, while a goal is something you take to be (very probably) worth doing. For instance:

(Real-world assumption) 'Outdoor leisure users often get wet.'

(Conclusion/assumption) 'Outdoor leisure devices will often get wet, too.'

(Project goal) 'Make the device waterproof.' (See Figure 7.10 on page 181).

7.4.4 Rationale Models

When one admits that nothing is certain, one must, I think, also admit that some things are much more nearly certain than others.

Bertrand Russell

A rationale model is a diagram of the reasons for something, such as a set of requirements. Modelling is appropriate when the justification for a requirement is contentious, with arguments on both sides.

Guest Box: Soren Lauesen on Business Goals

What follows is an extract from Lauesen's *Guide to Requirements* (2007), followed by the matching section of his requirements template [5]. Here, Lauesen is talking about business goals and how the system will meet them. First he traces the goals to the basic solution ideas. Next he traces the ideas to specific requirements. (The requirements themselves are not included in this extract.)

Extract from Guide to Requirements

B. High-level demands

B1. Business goals

This section of the template contains the business goals of the system, arranged in a table to show how the goals are to be met. The first column is the goal, the second the solution in broad terms, the third the requirements that make it possible. It is emphasized that the goals aren't requirements to the supplier, but background information. The last column allows the customer to state the deadline for meeting the goal. When stated, it is the deadline for the joint effort of the supplier and customer. The supplier should bear in mind that the customer also needs time for the organizational implementation.

The business goals serve several purposes:

- a. They tell the supplier what the customer wants to achieve.
- b. They are important criteria for choosing a solution.
- c. They help the customer check that the crucial requirements are included.

In the example, goal 1 (efficient support of all user tasks) is a very broad goal that depends on a lot of requirements. It is stated in such a way that it allows the customer to discard solutions that poorly support one or more tasks. As an example, the surgeon needs a good overview of the patient's situation in order to make the right decision. It must be possible

to discard a system with a poor overview screen although this is just one of 1000 details in the system. In public tenders, the rules are very strict. The business consequences may be severe if such a system cannot be discarded for legal reasons. Goal 1 ensures that it can.

In the example, the customer had identified goal 3, *continuous improvement of the work processes*. However, he hadn't realized that this required a new organizational structure – an *advisory board* – that should develop, test, and deploy new standard procedures for patient treatment. This required IT support, but the customer didn't realize it until the goal table had to be filled in.

Don't specify a lot of goals. If there are more than 10, check that they are not just requirements. We often see 'goals' of this kind: *It must be easy to print consumption reports*. Although this was important to one of the stakeholders, it is a simple system requirement, not a business goal. A business goal is about the results of the entire organization, not just something the computer can do.

If you cannot write something reasonable in column 2, it may be a sign that the goal is not a true business goal, but a requirement. As an example, if the goal is: *It must be easy to print consumption reports*, it will be hard to write a large scale solution. If you cannot write a real goal, simply leave column 2 blank.

Measuring the goals. A really good goal can be measured and compared against the existing state of affairs. Goal 2 is clearly of this kind. Goal 1 can be measured on a subjective scale of degrees (e.g. 1 to 5), or as the number of tasks performed per person per day. Goal 4 could be measured as operational costs before and after system deployment. Although the goals can be measured, the customer need not reveal the measurements. They might tell the supplier which price the customer is willing to pay.

Extract from Template

B. High-level demands

This chapter explains how the customer's business goals are met through the requirements and how to mitigate high-risk requirements.

B1. Business goals

The customer's reason to acquire the system is to reach some business goals. The customer expects that the system contributes to the goals as stated below. The supplier can rarely meet the goals alone. Customer

contribution is needed too. This means that the goals are /it not requirements to the supplier. They are shown in a table only to provide overview.

All goals are important and the sooner they can be met, the better. Some goals are crucial to meet at a specific date, for instance for business or legal reasons. Such deadlines are shown in the table.

Goals for the new EHR-system	Large scale solution	Related requirements	Deadline
1. Efficient support of all user tasks.	<i>All relevant data are available during the task, and all parties can see the health record.</i>	<i>Support for all tasks in Chapter C and all data in Chapter D.</i> <i>Adequate response times in L1.</i>	
2. Reduce medication errors from 10% to 20%.	<i>Avoid manual steps – record the prescription immediately.</i> <i>The system checks for validity, drug interaction, etc.</i>	<i>Support for task C10 (clinical session), in particular subtask 2 (assess the state of the patient).</i> <i>Support for task C11 (prescription), almost all the subtasks.</i>	
3. Continuous improvement of the work processes.	<i>Easy to set up and modify standard treatment plans.</i> <i>Easy to integrate the system with new lab systems, etc.</i>	<i>Support for task C80 (advisory board).</i> <i>Requirements in sections E3 and F10 (system expansion and integration with new systems).</i>	
4. Lower operational costs.	<i>Replace several expertise-demanding systems with one.</i>	<i>Support for all tasks from the previously separate work areas.</i>	
5. Meet the new EU rules on	1/1-2008

Reproduced with permission of Soren Lauesen.

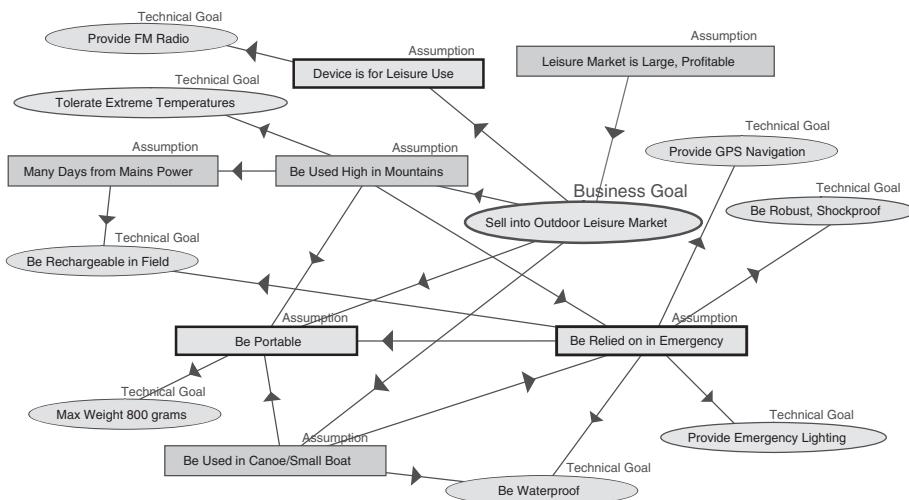


Figure 7.10: A rationale model consisting only of assumptions and goals.

A simple kind of rationale model (Figure 7.10) that we find helpful represents reasons only as assumptions (for an explanation of why, see the box, ‘Plato 0, Toulmin 1’).

Assumptions are drawn as boxes; requirement/design conclusions, which are essentially project goals, are drawn as bubbles.¹

Key goals and assumptions can conveniently be emphasised with a thick border or brighter colours.

Arrows are drawn to show how assumptions support each other and ultimately the argument’s conclusions – the goals.

You may find it interesting to compare Figure 7.10 with the goal model in Figure 6.1 at the start of Chapter 6. The rationale model, by showing explicitly how the goals are derived, arguably provides a better basis for identifying what the product should actually do, and what the priorities (see Chapter 10) for the project should therefore be.

You can also draw negative arrows (as with goal models) to show arguments that tend to weaken or contradict other assumptions (Figure 7.12).

In the example shown in Figure 7.10, the rationale model describes the multifunction device project that was discussed in Section 7.3.3, Rationalising a Set of Requirements. In that case, you are making informed guesses (assumptions) to help understand the reasons for the project and the individual requirements. If those guesses are confirmed by the project’s stakeholders, they must have been project goals all along. Therefore, the assumptions in this particular case are mostly early guesses at the larger, higher-level goals of the project itself.

¹This choice matches the UML convention for drawing use cases as bubbles, but use case goals are always functional (see Chapter 5).

Once you have confirmed them with stakeholders (perhaps by market survey), it makes sense to represent them explicitly as goals.

Note that traces, such as the arrows in Figure 7.10, can usefully be followed in either direction:

- back to where they came from, to justify a requirement;
- forwards from a goal, to discover requirements.

Note the difference between assumptions and goals. Goals say what you want (in future); assumptions say what you believe will happen anyway.

For example, a typical assumption in Figure 7.10 is that ‘the leisure market is large and profitable’. This is a belief about the world outside, which the project depends on and which could turn out to be true or false. The project must assume that there is a worthwhile market for its product – if not, it should be halted immediately. Making that assumption explicit is useful, as it shows management a key risk to the project. For example, if the economy goes into recession, the project’s funding ought to be reconsidered.

Tips for Documenting Rationale

- Keep it simple.
- Only provide rationale when it is needed, e.g. for disputed decisions.
- Choose notation that people on the project are comfortable with.
- Only record good, strong reasons (for and against).

7.4.5 The Goal Structuring Notation (GSN)

It is ironic that, while goals and assumptions are central to every project, they are probably most widely used in one particular system engineering discipline: safety, where they are just part of the conceptual toolkit that safety engineers use to argue that systems will be safe. The ways that safety people use goals and reasoning may provide useful hints for creating requirements.

GSN is one of several visual languages used in safety engineering. It explicitly uses goals, along with arguments (pieces of reasoning) and pieces of factual evidence (Figure 7.11).

A GSN diagram starts with a single goal (a box). The goal is generally to argue that a system is safe to use. The complete set of arguments and evidence, often a weighty pile of documentation, forms a ‘safety case’, which is presented to the safety regulator. If the regulator believes the safety case, permission is given to operate the system.

The top-level goal is broken down into a set of other diagram elements, which can be (sub)goals, arguments (parallelograms, called ‘strategies’), pieces

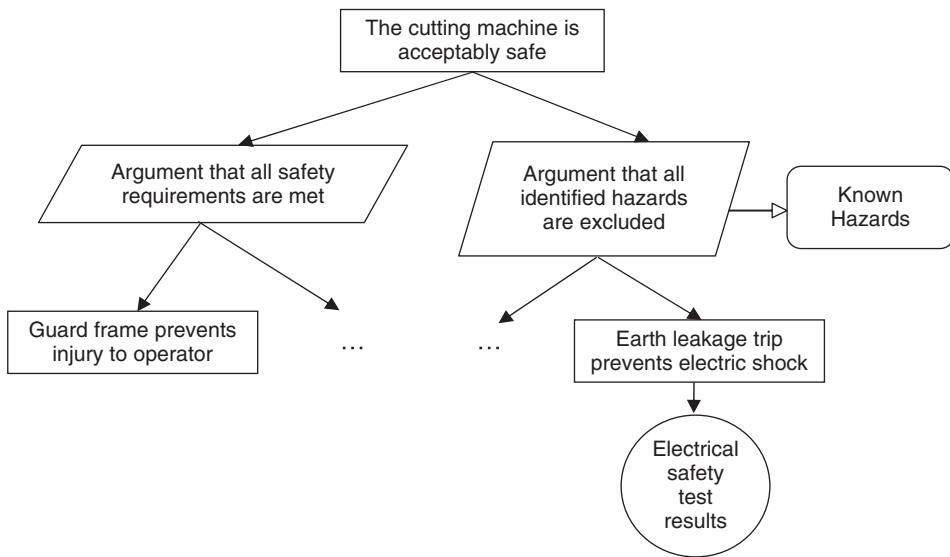


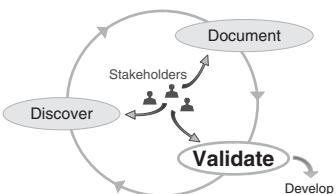
Figure 7.11: A GSN diagram expressing a safety rationale.

of evidence (circles, called ‘solutions’), and context statements (rounded rectangles). A goal or other element is linked by arrows to all the subgoals or other elements that it is broken down into (i.e. the arrows go down from main goal to subgoal, the opposite of ordinary goal diagrams).

More than one goal can be supported by a given piece of evidence; several pieces of evidence can support one goal. Suitable evidence could be an analysis of the faults that could cause a particular hazard, a state model for a system, or a set of test results.

There are several other notations for defining the reasoning in safety cases that are broadly similar to the GSN. These include, for example, claims-arguments-evidence (CAE). Tool support is discussed in Appendix C.

7.5 Validating Rationale and Assumptions



The way you’ll need to validate your requirements’ rationale depends on how you have documented it.

Formal (mathematical) completeness, correctness and consistency proof can be applied to classical reasoning but not to practical Toulmin-style reasoning (see Section 7.4.4). Therefore, for most purposes, you should not hope to prove consistency. Instead, a rationale is:

- complete if all serious and relevant arguments are documented (e.g. as assumptions);
- correct if people broadly agree with it;
- consistent if it defines the arguments for and against fairly, and reasonable people can then agree with the conclusions.

We do not recommend that you create a rationale model for every requirement, so a complete set of models means covering all contentious requirements adequately.

7.5.1 Rationale Walkthrough

If you document your rationale as a text attribute or column beside your requirements, then you should walk through the requirements, checking the rationale of each one as you go.

An effective format for a rationale walkthrough is a workshop in which the author of each requirement briefly explains what is intended and why, and the other participants ask any questions about the strength and persuasiveness of the arguments and evidence for the requirement.

The outcome may be to accept or reject the requirement (e.g. to change its status to ‘approved’, ‘rejected’, or ‘in rework’), to set its priority (e.g. to ‘essential’) or to revisit the justification text, the requirement text, or both.

7.5.2 Analysis of Traceability

If you document your rationale as separate statements traced to requirements, then you need to check requirements and rationale together, ideally using a traceability view in a requirements tool (or an equivalent hypertext or exported table). Again, a walkthrough workshop is likely to be effective.

A key point is that if a requirement has no trace to a justification statement, it is clearly not formally justified; but if there is such a trace, that does not show that the requirement is adequately justified, even if the traced justification is correct and relevant. You have to use your engineering judgement to decide if the evidence is sufficient to justify including the requirement.

If you use traces to goals as the main part of your justification approach, then you need to check that those traces are correct and sufficient to justify the requirements. This means that your review has to step through the requirements, looking at:

- explicit rationale statements and diagrams (where these occur);

- the traced-to goals.

The purpose of the review is to satisfy the team that the goals and rationale:

- are clear and accurate, and explain what the problem is;
- are fully met by the requirements;
- could not be met by fewer or simpler requirements.

It may take several assumptions, goals or other items to justify a requirement. In the example analysed in Section 7.3.4, Inverting Risks, the requirements for insulation and processor heating had to be justified by a system requirement (the operating temperature range), which in turn was justified by two assumptions (Figure 7.12).

Warrant and Rebuttal

Controversy can often be calmed by documenting counter-arguments or negative rationales. Toulmin [2] calls these ‘rebuttals’. A simple approach is to draw a line across the middle of a rationale diagram. Model the warrant above the line and the rebuttal below, as illustrated in Figure 7.12. This shows everyone that their arguments have been heard, so they don’t have to repeat themselves. The basis for a decision – or things that need to be investigated further – will then often be clear from the diagram.

Take care not to clutter rationale diagrams with unnecessary detail. Focus on the essential arguments. Do not, for example, put in trivial assumptions just to provide something on the ‘rebuttal’ side.

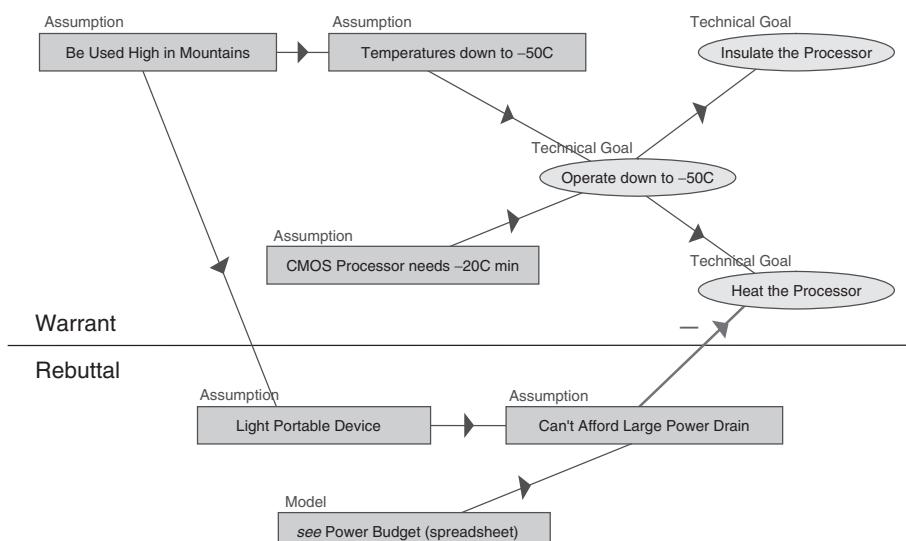


Figure 7.12: Warrant and rebuttal: Justifying protecting a processor from the cold.

Decision Support

Contentious design decisions may need to be justified by a whole rationale model to themselves, just as safety issues may need a whole safety argument. It may be worth preparing a separate rationale diagram for each such ‘hot’ issue.

A diagram clarifies the basis for a decision. The actual decision should remain a matter for informed human judgement. The existence of a possible argument for or against does not show that something is right or wrong; it just suggests a possibility.

Where you have a quantitative model or body of evidence for an assumption, you can provide a placeholder box for it on the rationale model, labelled as a hyperlink (e.g. ‘See Risk Model’; or ‘See Power Budget’, as in Figure 7.12). Many tools allow you to insert actual hyperlinks to connect diagrams to other documents in this way.

Can you automate decision support?

People have often tried to automate decision support by putting numbers (weights) on rationale statements and connections (traces) between them. They then use an algorithm (such as the sigmoid function used in neural nets) to propagate the weights through the network of arguments like those in Figure 7.12. If there are several possible outcomes, each will end up with a different score and, in theory, you then take the one with the highest score as the winner.

So far, such automated decision support has only really proved suitable for very well-defined (repetitive) decisions. In contrast, decisions on requirements and design issues are generally unique, and may have to be made quickly on quite limited evidence.

7.5.3 Things To Check Rationale and Assumptions Against

- Your organisation’s mission and its business objectives
- Your project’s original brief (vision, objectives, etc; Chapter 3)
- Goals and requirements that rationale models and statements are traced to (is the traceability complete? Is it correct?)
- Scenarios (Chapter 5)
- Definitions (Chapter 8)

- Reasons stated by stakeholders in interviews (Chapter 11), etc
- Trade-off decisions (Chapter 14)

7.6 The Bare Minimum of Rationale and Assumptions

- Make sure you know why you're doing what you're doing.
- Make sure the key assumptions on which your project depends are stated explicitly.
- Write a few words of rationale beside each requirement.
- Trace 'system' (product) requirements back to your project's goals.

7.7 Next Steps

- Provide measurements for your requirements (Chapter 9).
- Prioritise your requirements (Chapter 10).

7.8 Exercise

Choose a requirement for a restaurant's IT system, or another system that interests you, which you think can be argued strongly for and against. Draw a warrant and rebuttal-style rationale model (like Figure 7.12) to illustrate the arguments.

Hint: Take care to focus on the essence of the problem, and not to clutter the diagram with minor details.

7.9 Further Reading

7.9.1 Discovering Assumptions

1. Dewar, J. (2002) *Assumption-Based Planning: a Tool for Reducing Avoidable Surprises*, Cambridge: Cambridge University Press.

Dewar's is one of the few books to major usefully on assumptions, and fortunately it is a really good one. The book describes many simple and effective techniques for discovering assumptions – the reasons behind decisions. While it is designed to find the reasons for plans, it works very well on requirements too.

7.9.2 Reasoning

2. Toulmin, S. (1958) *The Uses of Argument*, Cambridge: Cambridge University Press.

Classical reasoning works only from premisses that are accepted as true. Toulmin takes a giant step towards reality by describing how to argue from premisses which are probably sound, but which may fray somewhat round the edges. The book is written as philosophy, but it is so simple and clear that it remains an excellent introduction to the subject that Toulmin effectively created.

7.9.3 Modelling Rationale

3. Kirschner, P.A., Buckingham Shum, S.J. and Carr, C.S. (Eds) (2003) *Visualizing Argumentation*, London: Springer.
4. Burge, J.E., Carroll, J.M., McCall, R. and Mistrík, I. (2008) *Rationale-Based Software Engineering*, Berlin: Springer.

These are two somewhat academic books on rationale. Kirschner is an edited collection of chapters on ways of presenting rationale graphically. Burge is a text on the many research approaches that try to put rationale into service in software development.

7.9.4 Tracing to Goals

5. Lauesen, S. (2007) *Guide to Requirements SL-07: Template with Examples*, Copenhagen: Lauesen Publishing.
See the Further Reading section in Chapter 6 for comments on Lauesen.
The template can be downloaded from: <http://www.itu.dk/people/slauesen/Papers/RequirementsSL-07.doc>

7.9.5 Goal Structuring Notation (GSN)

6. Kelly, T. and Weaver, R. (2004) *The Goal Structuring Notation – A Safety Argument Notation*, <http://www-users.cs.york.ac.uk/~tpk/dsn2004.pdf>
Tim Kelly, the inventor of GSN, describes the notation.

7.9.6 Satisfaction Arguments

7. Hull, E., Jackson, K. and Dick, J. (2005) *Requirements Engineering*, 2nd Edition, London: Springer.
In chapter seven of this short industrial textbook, Jeremy Dick describes his take on satisfaction arguments and rich traceability.

CHAPTER EIGHT

Definitions

'When I use a word,' Humpty Dumpty said, in rather a scornful tone, 'it means just what I choose it to mean – neither more nor less.'

Lewis Carroll, Through the Looking-Glass

Requirement Elements	Definitions	Priorities	Measurements
Discovery Contexts	Rationale and Assumptions		
Introduction	Qualities and Constraints		
From Individuals	Scenarios		
From Groups			
From Things			
Trade-Offs			
Putting it all Together	Goals		
	Stakeholders		

Answering the questions:

- What does this mean?
- Which roles are involved?

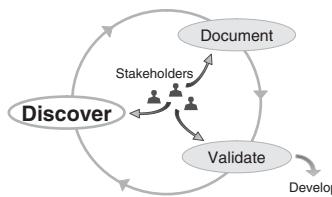
... so people talk the same language and use terms the same way
... so that people agree, and understand the same things by the same words.

8.1 Summary

Agreed definitions of acronyms, terms, roles and data save confusion, clarify the meaning of requirements and can resolve longstanding disagreements.

A solidly reliable set of definitions of terms enables requirements to be written more simply, because issues of meaning can be handled just by using terms which are already defined and agreed.

8.2 Discovering Definitions



A large part of the role of a requirements document is to explain: to explain what is wanted; to say why it is wanted; and, not least, to explain the problem that is to be solved. That demands not only a set of requirements but also a set of well-chosen terms with precise meanings – in other words, a dictionary.

An accurate project dictionary prevents many kinds of error, and strengthens the requirements by clarifying their meaning.

In some cases, as with definitions of data items, dictionary entries are, effectively, requirements: ‘customer surname’ has to be a 40-character string, and so on. This is not the sort of thing you will usually define on the first day of a project, so the project dictionary will take time to create.

Unlike with requirement elements like goals and scenarios, we don’t imagine you will hold a definitions workshop on your project – at least, not normally. The work of discovering terms that need to be defined is usually a background activity. As you’re interviewing, you notice a word being used in a special

way, and ask about it. While you're running a requirements workshop, you notice an issue about the meaning of a word, and check that a definition is agreed.

Can't we use the corporate glossary and acronym list?

On hearing that a project is to have a dictionary, organisations quite often respond that they already have a corporate glossary or 'jargon-buster', or perhaps a list of acronyms.

Neither of these is enough to fulfil the role of project dictionary.

- The corporate glossary is usually very general, and contains many terms that are not helpful to the project. It usually omits most of the terms that the project needs, especially the more technical ones.
- The corporate list of acronyms is, again, general to the whole business. It also tends to contain many obsolete acronyms; even if it is carefully kept up to date by the addition of new acronyms, old ones are rarely removed. This means it is often large and confusing, rather than helpful to the project.

However, you need to be aware of the corporate definitions, so that differences from them are clear and explicit.

The work of discovering project dictionary terms may suddenly come to the foreground in, say, a scenarios workshop when people disagree on what something means, or what should happen. Quite often, you can resolve that issue by finding out what people do mean by a term, and creating new definitions if necessary.

8.2.1 Synonyms

Synonyms are terms that have the same meaning, or overlapping meanings. Synonyms are dangerous because of the confusion they can cause. They can lead directly to specification errors, most easily if a role or data item is referred to in two ways.

For example, suppose 'upper limit of tank pressure' and 'max tank pressure' denote the same thing, but are defined separately. Then, if one of them is redefined in the data dictionary, different requirements may call for inconsistent and possibly dangerous behaviour.

'Equivalent'

A classic dictionary experience happened to Ian in a workshop on a safety-related system upgrade project. A discussion sprang up about 'equivalence'.

One view was that to be equivalent to the existing system, the upgrade had to stay with the traditional electronics, which meant big single components like capacitors and inductors, soldered via big thick wires onto big thick circuit boards. The boards had to be of the same design as those for the old system. Equivalence was essential to avoid having to start all over again with a new safety case for the entire system. That would have been impossibly slow and costly.

The other view was that you could use new components, microprocessors and software, as long as the result was to do the same decision-making as the old system. In fact, you could take the old system's behaviour as the minimum, fall-back case. You could do whatever you liked to make the new system work better, as long as you could, without fail, go back to the old safe way of doing things if any problem arose.

The discussion rapidly became heated.

The discussion, we realised, was about the critical issue of how much freedom the project had to make use of new technology, such as software and microprocessors. One side felt that equivalence could only be preserved by extremely conservative design. The other felt that equivalence was something much looser.

We asked both sides what they meant by 'equivalent', and wrote their definitions on the whiteboard. It was an 'Aha!' moment.

'There are two different sorts of "equivalence" here. How would it be if we called them "physical equivalence" and "functional equivalence"?' we asked.

We wrote these two titles on the board, and attached them to the two definitions. That made it possible to ask the central question in the debate:

'Do we know if the new system has to be physically equivalent to the old one, or would being functionally equivalent be allowable?'

It took some time to find the answer – functional equivalence was fine – but the heat had already gone out of the discussion. Everyone was happy.

Multiple definitions can appear when people working on requirements find they need to explain or define something in different contexts. They may separately invent different terms or use different phrases to denote the same thing.

For instance, you might find informal phrases (for example, 'the ordinary user', 'office users', 'authorised operators' and 'the trained operator') in use

in different scenarios, all meaning approximately (but not exactly) the same group of people. Some of these terms might then be defined in the project dictionary. Note that synonyms are dangerous regardless of whether they are scattered through the requirements or in the dictionary.

Removing Synonyms

The correct solution is for the project to:

1. select the term it is most comfortable with;
2. replace all the informal phrases in the requirements (and the dictionary) with the chosen term;
3. define the term in the dictionary.

Avoiding Synonyms with Mark Up

You may find it helpful to mark up dictionary terms wherever they are used, e.g. by putting them in italics when you use them in requirements:

‘When *Current_Tank_Pressure* is greater than or equal *Max_Tank_Pressure*, then . . .’

This has several advantages:

- it reduces the risk of people editing names of terms by mistake;
- it makes the terms stand out, which:
 - reminds analysts of terms already in use, reducing the risk of creating synonyms
 - helps developers and programmers to find data dictionary items;
- it makes it easy for a script to search for terms and check them against the dictionary.¹

8.2.2 Homonyms

Homonyms are terms that have multiple meanings. They are inherently ambiguous, and can cause confusion. For example, ‘manager’ can mean both a human operational role, and a software object.

The main danger with homonyms is that people from different domains may be so familiar with one meaning of a term that they do not realise that

¹For example, the free dictionary tool on www.scenarioplus.org.uk searches a DOORS module for terms marked up in italics, links them to their definitions in the dictionary module, and creates new (empty) definitions as needed. Tools are discussed in Appendix C.

Table 8.1: Definitions of ‘service’ in different domains.

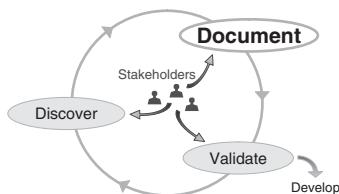
Domain	Meaning of ‘service’
Architecture	Pipe or cable coming through a wall into a building to supply gas, water, telephone etc, e.g. service duct, service cabinet
General public	Quality of attention and care received from a business
Business	Work provided in return for money; this kind of service is an alternative to buying a product and using it to do the work yourself
Railway	A timed train route, e.g. the 16:10 express service from Glasgow
Software	A piece of software, running on a network, that can be called up to perform a specified task, possibly for payment
Government	a. An arm of government, e.g. Health Service, Civil Service b. A force, e.g. the armed services, police service

people with other backgrounds and cultures use the term very differently. The most overloaded terms are often innocent-looking everyday words, such as ‘service’ (Table 8.1). There are at least 41 listed meanings of this word in good English dictionaries.

Confusion could possibly arise if, for example, a software engineer spoke of the use of *software* services to support the delivery of *business* services by a company to its customers. By the way, it is in the business sense that ‘services’ is used in the subtitle of this book; a more formal definition is given in the Glossary.

As another example, the word ‘signal’ has different meanings for traffic engineers, electrical engineers and software people. The simplest words can be the most slippery.

8.3 Constructing the Project Dictionary



An effective project dictionary may need several components, depending on the nature of the project:

- acronyms;
- definitions and designations;
- roles (operational stakeholders);
- data definitions.

Let us look at each of these in turn.

One Dictionary or Separate Lists?

Should you put all the acronyms, terms, roles and data items in a single, multipurpose project dictionary, or have a separate list for each purpose?

- A single dictionary makes it easy for everyone to remember where to look, especially if you can publish it on a project website on your organisation's intranet.
- Separate lists are more convenient for later analysis, for example if someone needs to make a table of permissions from your list of roles.
- If you use a database tool (or simply a spreadsheet like Excel), you can get the benefits of separate lists with the convenience of having just one: identify the type (acronym, role, etc) of each item; then select or filter the dictionary to show just the acronyms, etc.

8.3.1 Acronyms

Every project uses abbreviations and acronyms, just as every organisation does. These may seem very familiar to 'everyone', but in practice they form a forbidding barrier to several groups of people:

- new members of the project team;
- clients;
- suppliers.

You have surely seen documents where every page is full of acronyms. You probably found them hard to read, and you felt excluded. A good requirements document therefore avoids using too many acronyms, and carefully defines the ones it does use.

It is not absolutely forbidden to have an acronym with two different meanings in different contexts (like PM = program module and PM = project manager), but avoid it if you can.

8.3.2 Definitions and Designations

Populating the main part of the project dictionary might seem a simple matter:

- notice undefined terms;
- write suitable (project-specific) definitions.

But there is one distinction that you will probably need to make to achieve clarity and consistency in your dictionary. This is between references to things

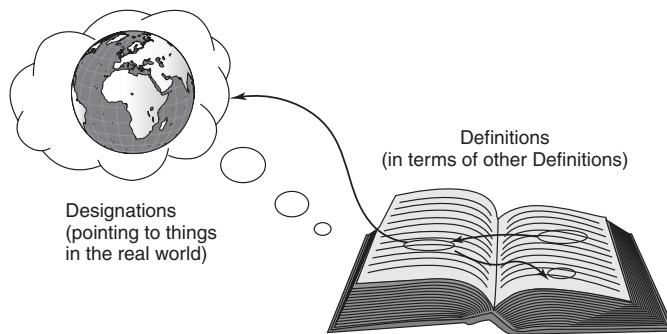


Figure 8.1: Definitions versus designations.

within your system – those under your control – and things in the world outside. The author of Jackson Structured Design, Michael Jackson (1995), calls these ‘definitions and designations’ [1] (Figure 8.1).

The Obvious Name?

Names, the uses of particular words and the concepts that go with them form a large part of the culture of an organisation. Sociologists define culture as the unspoken (tacit) assumptions that are shared by everyone in that culture. Crossing the divide between cultures, as between a client’s and a developer’s, is not easy.

Many years ago, on a project for a government-run organisation, we needed to find a name for the onscreen form that a customer had to fill in to order a specialised image. The customer could view small, thumbnail images via a desperately slow modem, select the ones they wanted, and order the desired quantity to be printed out as enormous glossy photographs. It was, though we didn’t know it, a very early online shopping system for a technical clientele, long before Amazon or the Web were born or thought of.

We met the client.

‘You can’t call that a form,’ he said. ‘A form is a piece of paper.’

We ended up calling it a ‘simulated form’ throughout the user documentation.

Definitions

Within reason, you are free to choose any names you like for things within your system. Of course, there are good, clear, helpful names and poor, ambiguous ones. It is hard to offer much concrete advice here, but if you can find an

accurate and ‘intuitive’ match between a word or phrase in common use and the thing you need to define, then use it.

Be aware that, just because you and your colleagues in your organisation are familiar with a term, it does not mean that people in other organisations (such as your customers, suppliers or contractors) will know what it means. Using a common English word that has misleading associations in their minds could make matters worse. An example of a word that has different meanings to different groups of people is given in Table 8.1.

Anti-jargon

Sometimes, people who are very familiar with a domain do exactly the opposite of creating ever more complex jargon for domain elements.

You might expect, for example, that when a submarine commander wants to make his ship dive, he says: ‘Number 1, seal and verify the conning tower main hatch’. (‘Aye aye Captain!’) Not a bit of it. He says: ‘Close the lid’.

You’ll guess, therefore, that when customs officers decide to investigate a suspect shipping container, they say: ‘Let’s look in that box’. And when a television producer needs to refer to the finished high-definition video images the team is creating, she just says ‘pictures’.

These attractive simplicities of speech do not necessarily translate into terms you can put into the project dictionary. Most likely, you’ll need to ask which term to write down for formal use.

Designations

A designation is a pointer to things outside the system – the equivalent of a gesture during a spoken explanation. Surprisingly, perhaps, references to the real world are the vaguest part of the project dictionary. (The philosophically inclined will recognise here an echo of Plato’s description of the cave dwellers who could only see dim shadows projected from the world outside.) The best you can do is to say enough to awaken recognition of concepts that readers already have. Then, they can share an understanding of what you are referring to.

For example, this book’s Glossary designates a ‘domain’ as: ‘A realm of business discourse’, and goes on to give some examples of domains. The designation depends on the reader’s familiarity with the words used, or at least with the examples.

Definitions, in contrast, have an exact meaning in terms of their relation to other parts of the system: this is part of that; this one is a subtype of that one; this is an operation on that. Definitions must all fit together.

For example, this book's Glossary defines 'prioritisation' as the 'Process of assigning **priorities** to **requirements** or **features**, with the purpose of **triage**'. The terms in bold are references to other terms defined in the Glossary.

Constructing the Main Dictionary

We are now ready to state how to build the main part of the project dictionary.

The basic rules for definitions are to:

- define each term, as far as possible, by reference to other terms in the dictionary (the Glossary of this book is built in that way);
- avoid synonyms and overlaps of meaning.

Why is that? Because each time a definition makes use of an undefined term, ambiguity and risk are created. Either the dictionary writer or, worse, a requirement writer, will be tempted to put in a small local explanation to clarify the term that is missing from the dictionary. That means that very long requirements or dictionary entries are signs of possible trouble.

So, paradoxical as it may sound, definitions should be internal and seemingly circular. Isn't that harmful? No, because the circularity is broken by the bare minimum of designations – as long as the reader has enough shared knowledge of the domain and culture being referred to.² You can help by providing thorough context (Chapter 4) and rationale (Chapter 7).

In contrast to the rule for definitions, synonyms in designations may be unavoidable. Since designations refer to things in the world outside the product or system, your project may have little control of the terms people use as designations. The best you can hope for is to clarify the meanings in common use, and to point out possible confusions.

Tips for your Project Dictionary

- Watch out for signs of confusion or disagreement in workshops – it often means that a term has two meanings.
- As you start to collect information about a project, keep your own list of terms, and watch out for name clashes.
- Ask people to help you with the meanings of terms.
- Ask who would be the best person to explain a term.
- In interviews, ask people about terms that they mention.
- In workshops, get a volunteer to make a list of terms that are discussed.

²See the box 'Collaboration at a Distance' in Chapter 12.

8.3.3 Roles (Operational Stakeholders)

Chapter 2 looked at identifying the full range of stakeholders in a system. They play many roles. The ones that usually matter most for the project dictionary are the operational roles, for people involved in day-to-day operations of the business and the product under design.

Example: Roles in a Retail System

Consider a retail system in a department store. The system includes a central database server, which is connected to point-of-sale devices like cash registers and card readers in each department.

These are usually operated by their normal operators (see Chapter 2), **retail sales assistants**.

However, if there is a problem with a transaction (say, a card payment is refused by a customer's bank) or if an assistant is taken ill, then a **retail sales manager** may conduct retail sales directly and operate the devices in person.

Still other roles involved in the business, such as **stock controller**, **internal auditor** and **maintenance technician**, may also operate the retail system for their own purposes.

Each of these roles needs access to the system, but should be limited to tasks required for their role. For example, it would be appropriate for a maintenance technician to test the operation of a cash register by running a dummy sales transaction, but inappropriate for that technician actually to sell any goods.

Roles and Responsibilities

Early in requirements discovery, the list of transaction types (scenarios) is known only approximately and, indeed, the list of roles may be only partially defined.

You can start by defining roughly what responsibilities you expect from each role, by writing a general description of the role in the project dictionary. This shows both what is understood about the system and, more importantly (with a 'to be decided' (TBD) note against the sketched list of transactions), what is not yet known (Table 8.2).

Table 8.2: Role definition in Project Dictionary.

Term	Type	Definition
Retail sales assistant	Role	Works in a department, at a sales counter. Operates point-of-sale devices to sell goods. Transactions: sale of goods. TBD
...

The requirements for the retail system need to say who may operate what. This will ultimately mean the specification of responsibilities (an exhaustive list of transactions permitted for each role), once the design is fully known (Table 8.3).

Nowadays, responsibilities are increasingly defined in software. In the retail system, that means they will be managed via the database; for instance, perhaps a permissions service running on the department store's intranet will be accessed by all transaction applications.

However, not all business operations are (fully) automated, so at least some aspects of roles and responsibilities remain manually controlled. For example, retail sales managers will ensure that retail sales assistants do not leave their sales counters unattended, by keeping an eye on them. Operational procedures need to be defined in a similar way (e.g. with scenarios or flowcharts), whether or not they are to be automated with software. Indeed, you may not know until much later whether a procedure is to be automated or not. As shown in Table 8.3, you can record nonautomated responsibilities (and operational procedures) explicitly as 'Manual'.

Table 8.3: Roles and responsibilities (R&R) table.

Role	Responsibilities '{xyz}' means 'can take on any of the responsibilities in role xyz'	Implementation
Retail sales assistant	Customer welcome and inquiries	Manual
	Sale of goods	Automated
	Return of goods	Automated
	Pricing inquiry	Automated
Retail sales manager	{Retail sales assistant}	{see that role}
	Department sales summary	Automated
	Retail sales staff supervision	Manual

Stock controller	{Retail sales manager}	{see that role}
	Stock report	Automated
	Stock correction	Automated
Internal auditor	{Stock controller}	{see that role}
	Audit report	Automated

Maintenance technician	Dummy transaction	Automated
	Diagnostics	Automated
	Report fault	Automated
	Clear fault	Automated
...

8.3.4 Data Definitions

Like events (see Chapter 4), data form an important part of a system's interfaces, both with other systems and with human operators. Interface definitions (the data sent or received for different events) may fix a large part of a product's design. Data definitions – traditionally known as the data dictionary – are therefore a vital part of the requirements.

Sources of data definitions include:

- **standards**, for published interface protocols, e.g. in banking, telecommunications;
- **existing systems and products**, for common interfaces (e.g. to minimise retraining of human operators, for commonality or interoperability of products), especially in product lines, but also for upgrading custom systems;
- **prototyping** (see Chapter 13), to explore new user interfaces, e.g. for interactive software and consumer products;
- **modelling** of the data itself, iterated with interviews and workshops to validate the models, correct misunderstandings and fill in gaps;
- **analysis of algorithms**, to determine required input parameters, e.g. in embedded systems and scientific programming.

These are very diverse sources. You may have to use any or all of them on your project, depending on its type.

The traditional approach is to build a data dictionary and to cross-check it against other models, especially an Entity Relationship (ER) model of the data. In essence that hasn't changed, though the data model is nowadays usually a UML class model [3] conveying the same information. In that framework, data are represented as a combination of attributes of classes and relationships between classes (Figure 8.2). However, Lauesen [4] observes that ER models can be drawn to take up less space than UML class models, and to be easier to read. Data modelling is a skill in itself, covered in many textbooks such as Maciaszek's [2].

As usual with modelling, there is a danger of diving too deeply into design, making premature decisions before the intent of the data handling is fully understood. Simply naming a class tends to influence database design, for instance. Make it clear if your classes are just suggestions, i.e. you are not requiring that a database table should exactly correspond to each class. A modern way of doing this is to define 'business objects' rather than explicit database elements. This leaves the actual implementation choices open for the software designers to make.

It may be very valuable to have a period of intentionally playful exploration of user interface prototypes (see Chapter 13), simply to get a better idea of what data you need to define.

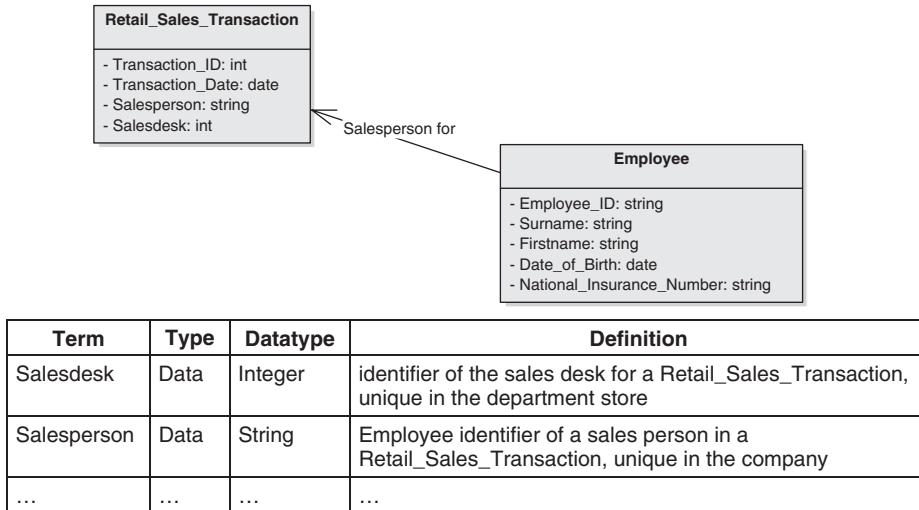


Figure 8.2: Fragment of a data model and some associated data dictionary entries.

8.3.5 Constraints as Data

*Much of what you are doing
in drawing a class diagram
is indicating constraints.*

Martin Fowler

You have already seen that definitions can carry a significant part of the burden of making requirements clear. A data model can also express several kinds of requirement directly (Figure 8.3).

The most salient feature of this class diagram is the relationship of the customer and the preferred medium. The note-style UML constraint, written inside braces {}, demands that:

‘The customer must supply an email address if their preferred medium is email.’

This is certainly a requirement; it makes no sense to allow email to be chosen as the medium if the customer doesn’t want to share their email address with the business. There are no restrictions in UML on how constraints are written; we can use English, or equations, or (as here) pseudocode, as long as the result will be correctly understood.

However, this explicit constraint is only one of several requirements documented in Figure 8.3. You may find the others less obvious, less interesting,

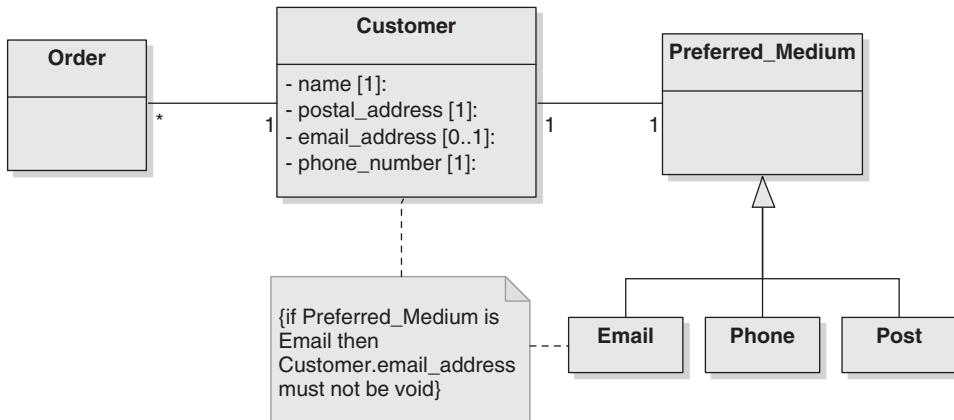


Figure 8.3: Modelling data relationships and constraints.

even trivial. But that does not mean they are easy to discover, or that they are unimportant. Here are some of the constraints ‘hiding’ in the diagram:

- The association line between Order and Customer insists that:
 - ‘each order must be placed by a single known customer’ (cardinality is ‘1’);
 - ‘a customer may place any number of orders’ (cardinality is ‘*’).
- The attributes of the Customer class insist that:
 - ‘every customer must have exactly one name, exactly one postal address, exactly one phone number and, at most, one email address’.
- The subtyping of the Preferred_Medium class insists that:
 - ‘the customer’s preferred medium must be exactly one of email, phone or post: no other values are allowed’.

Even this very brief excursion into data modelling is enough to show that a UML class model can represent requirements in several ways (Table 8.4). Some people call constraints of all these kinds ‘business rules’.

Table 8.4: Ways of representing requirements in a class model.

Explicit constraints	<ul style="list-style-type: none"> • Rules written in braces {}
Implicit constraints	<ul style="list-style-type: none"> • Associations between classes • Attributes of a class • Subtypes of a class

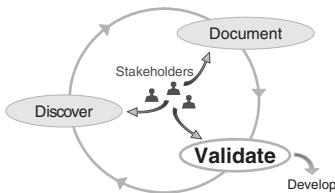
Fowler is thus certainly correct when he says that expressing constraints is a large part of the work of building class models. There are often alternative ways of representing the same thing and, as Fowler (2004) also says:

'The choice is much more about emphasis than about any underlying meaning' [3].

In Figure 8.3, for instance, the alternative options for Preferred_Medium have been emphasised for explanatory purposes.

Many other kinds of model can be useful for representing requirements precisely for programmers to follow, but most of them are quite dangerous for requirements discovery because stakeholders may not understand them fully. For instance, they may misinterpret them or claim they understand them when they don't. Or, they may grasp a few salient points but overlook important details. Basic flowcharts are widely understood; more subtle kinds of model – by which we include data models – are not. Care is needed when representing requirements in data models, because the majority of the requirements are essentially implicit. Skilled analysts and programmers will immediately understand the implications of a data model; do not assume that other stakeholders will. It is your responsibility to explain implied constraints to your stakeholders, for instance in a workshop.

8.4 Validating the Project Dictionary



When validating the project dictionary, you need to look for errors, omissions, and inconsistencies. These are handled in different ways (Table 8.5).

Make Haste, Slowly

It takes time to build a good, reliable project dictionary. The task will involve the whole project team, but it helps enormously to have a single person on the team who can energetically and systematically resolve any dictionary issues. The right person for the role will be patient, thorough, and interested in getting things right.

As with rationale, it is not essential to have traces to connect dictionary entries to the places where they are used – but it helps. If your project has a requirements database, you can construct such links semi-automatically, and

Table 8.5: Validating the project dictionary.

Dictionary problem	Solution
Errors	Get stakeholders in the team to check definitions in their areas.
Omissions	Check that terms, acronyms, roles and data items mentioned in other places (context models, scenarios, etc) are defined in the dictionary. Listen for 'new' terms in workshops, etc.
Inconsistencies	Read through the dictionary. Note any possible inconsistencies (often marked by synonyms or homonyms), and check with relevant stakeholders. If need be, propose new terms and their definitions to resolve inconsistencies, and get these agreed by stakeholders. Get an experienced data modeller on your team.

then view definitions alongside your requirements. This makes it easier both to understand the requirements and to validate the dictionary.

8.4.1 Validating Data Models

You may need to challenge the numbers (cardinalities) shown on association lines in data models (see Figure 8.3 and the discussion around it):

- Is it really necessary for a customer to specify a preferred medium?
- Couldn't there be a default, or is it easier to treat the default as if the customer had specified it anyway?

The answers to such questions may help to prevent odd or unexpected behaviour from software products.

A time-honoured guideline is therefore:

- Use any type of model or notation that helps you to express the requirements clearly and unambiguously.
- Then, find a way to play back the requirements to the stakeholders, to ensure they understand and agree that the requirements are right.

Two possible ways of validating data models are:

- Translate data models into a text suitable for stakeholders to read. Put every data constraint into English, e.g. 'Every customer must have exactly one phone number'. You can derive a complete set of such constraints by marking each class, attribute and association on a data model once you have translated it into English; or you may be able to find tools that can translate data definitions into rather stiff English automatically.
- Run workshops to take stakeholders systematically through data models, using scenarios, acted scenes, examples and any other means to bring the data models to life.

8.4.2 Things To Check Definitions Against

- Other definitions
- The requirement elements where defined terms are used (Part I of this book)
- Scenarios (Chapter 5)
- Rationale (Chapter 7)

8.5 The Bare Minimum of Definitions

- Make sure people on your project mean the same things by the same words.

8.6 Next Steps

- Write acceptance criteria or quality-of-service measurements for your requirements (Chapter 9).
- Prioritise the requirements (Chapter 10).

8.7 Exercise

For a restaurant's IT system, or another system that interests you, construct a project dictionary of the main terms that need to be defined.

Hint: Remember to ensure that any terms you use to explain other terms are themselves defined in the dictionary.

8.8 Further Reading

8.8.1 Definitions and Designations

1. Jackson, M. (1995) *Software Requirements and Specifications, a lexicon of practice, principles and prejudices*, Harlow: Addison-Wesley.
Many of Michael Jackson's sharp and witty insights into requirements matters can be found in this book.

8.8.2 Data Modelling

2. Maciaszek, L.A. (2005) *Requirements Analysis and System Design*, Harlow: Addison-Wesley.
A good thorough book on data modelling.
3. Fowler, M. (2004) *UML Distilled*, 3rd Edition, Boston: Addison-Wesley.
A convenient short reference to the UML.
4. Lauesen, S. (2007) *Guide to Requirements SL-07: Template with Examples*, Copenhagen: Lauesen Publishing.
Examples of data definitions can be found in this book. The template can be downloaded from:
<http://www.itu.dk/people/slauesen/Papers/RequirementsSL-07.doc>

CHAPTER NINE

Measurements

There's no point in being exact about something if you don't even know what you're talking about.

John von Neumann

Requirement Elements	Measurements
Definitions	Priorities
Rationale and Assumptions	
Qualities and Constraints	
Scenarios	
Context, Interfaces, Scope	
Goals	
Trade-Offs	
Putting it all Together	
Stakeholders	

Answering the questions:

- How will you know that this requirement has been met satisfactorily?
- How can you measure delivery of what is required?
- How will you tell that the product or service does what it says?
- How will you know that your suppliers have done what you asked?
 - so you get what you wanted (or at least, what you asked for)
 - so you know how to show that it works
 - and you can plan your verification campaign.

9.1 Summary

Requirements are all about communicating desired results – from customers to developers and testers, in particular. Testers use requirements to develop sets of tests that will show, at the time of acceptance, that a product meets the requirements. To make requirements sufficiently clear for test purposes, they need to be associated with acceptance criteria – statements of how much the product has to do to meet the requirements. Acceptance criteria make requirements measurable.

But development of a product is only one way that a desired result can be obtained. An alternative route is to write requirements for a service. A service provider delivers results, more or less continuously, often over a period of many years, in return for payment. Almost any desired result, whether for a business or for personal consumption, can be delivered as a service. The quality of a service (QoS) needs to be measured repeatedly or continuously, throughout the life of that service. QoS measures partly replace acceptance criteria for desired qualities (see Chapter 6) when the requirements form part of a contract for service delivery.

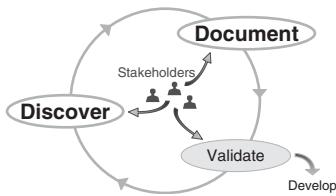
Some people believe requirements can always be written the same way; others believe that specifying services is entirely unlike specifying products.

In this chapter, we will look in turn at the discover-document-validate cycles for:

- acceptance criteria (to measure products);
- quality of service measures (to measure services).

We will see that there are many similarities between measurements for products and services, but some important differences.

9.2 Discovering and Documenting Acceptance Criteria



You say you want a \$100m project by next month. You have no stated acceptance criteria. I can deliver it!

Richard Stevens

Richard Stevens' joke is a terse and lively statement of the essence of measurement. If requirements are about asking for what you want, acceptance criteria are about ensuring that you get it, from your suppliers – very often, in the framework of a contract.

People have used many different names for this side of a requirement: measure of effectiveness (MoE), measure of performance (MoP), fit criterion, test criterion, postcondition and guarantee, as well as 'acceptance criterion' itself.

These names refer to slightly different measurements, but they all come down to the same thing: you say what to check for when the time comes to test, or otherwise verify, that the product does 'what it says on the tin' (Figure 9.1).

This does not mean that you have to plan all your tests in minute detail when you are writing your requirements. It does certainly mean that it is an excellent idea to get your test people involved in requirements work, and there is no better way than by getting them to help write effective acceptance criteria, or at least to check them.



Figure 9.1: Does it do what it says on the tin?

An acceptance criterion measures a single requirement. If you find you need two or more for a requirement, you may be able to make testing (and the enforcement of a contract) easier by splitting the requirement. Traditionally, a requirement is considered well written if you can identify a single test to show definitively whether the requirement has been met. With the move towards scenario-based testing, this is probably less important than it used to be, at least for functions.

With this understanding, let us consider how to discover acceptance criteria for different types of requirement. We will look in turn at functions and performance, scenarios and use cases, and a sample of qualities and constraints.

9.2.1 Acceptance Criteria for Behavioural Requirements

Behaviour may be specified by functions, performance requirements, scenarios/use cases, or a combination of these. Performance measures are tightly related to functions, so we will discuss them together here.

Functions and their Performance

The work of discovering acceptance criteria for a function is largely a matter of sharpening up the team's understanding of the requirement. After all, the way you know that a function has succeeded is to observe its output.

Let us take as an example the telematics system of a car. It is responsible for providing entertainment, navigation, traffic information and safety warnings to the driver and passengers.

You know you have succeeded in selecting a radio channel when you hear that channel playing:

Function: *The telematics system shall tune to a radio channel within three seconds.*

Acceptance criterion: *The correct channel starts playing within three seconds from selection of channel.*

Performance: Do You Mean that Quicker is Better?

Traditionally, stating a performance as 'within' means that a supplier should score more highly if he can promise to do better than the minimum value specified. '... tune ... within three seconds' suggests that the car maker would like the telematics supplier to tune the radio faster if possible – and may be more likely to award a contract if so. Naturally, this presupposes that there are several competing suppliers to choose from.

Conversely, when a requirement states a performance without using 'within', e.g. '... starts playing three seconds from ...' then there is no benefit from doing it faster.

If you find this too subtle, you should follow Soren Lauesen's (2007) approach and state explicitly, '[no] advantage to supplier for faster performance' [5].

When there is a performance that you care about very much, consider defining a cost function – a line or curve – relating the achieved value to benefit. In the case of a service, this should directly translate into the reward received by the service provider. In the case of a product to be paid for at acceptance, payment by performance is complex, and not often attempted.

The acceptance criterion for a function is closely tied to the function's performance, which is more of an attribute than a separate requirement. Indeed, choosing the right performance values is often the most difficult part of specifying product behaviour.

Notice that the acceptance criterion, specifying the required test result, effectively is the requirement, taken as a contract. You might conclude that, therefore, tests could replace requirements, but this is doubtful (Table 9.1). Tests do form an important part of the specifications in agile projects, but these are complemented by written scenarios in the form of 'user stories' (see box, 'Acceptance Tests in Agile Software Development'), and typically by hand-drawn models of system design, for example on a wall used by the group (see Section 12.4, Group Media in Chapter 12).

Acceptance Tests in Agile Software Development

In his thoughtful and carefully written book on agile software development [3], Mike Cohn (2004) advises programmers to write their acceptance tests 'in advance of coding the story'. Far from rushing straight into code, in the agile style, developers document their requirements as user stories – short scenarios (see Chapter 5). They then define acceptance criteria in the most concrete form possible: as working acceptance tests (coded test scripts). Since the user stories and the tests together form the specifications, they are written before the code.

This early focus on the desired results enormously clarifies the goal of each programming task. It enables development to proceed in small, clearly defined steps. Mistakes are detected quickly and are as quickly corrected. Practical results are achieved steadily. Each meeting with the 'users' has something more to demonstrate, and, therefore, an immediate opportunity to discover more requirements.

Table 9.1: Can tests replace measured requirements?

For	Against
<p><i>Write tests before coding.</i> (Mike Cohn)</p> <p>In ‘agile’ approaches, functional requirements are actually specified by writing tests (as well as user stories).</p> <p>Tests are practical and down to earth.</p> <p>Everyone can see test results.</p> <p>A focus on making programs run and deliver results, early, forces development to iterate between requirements and design (Chapter 14). The client can then see early working versions as prototypes (Chapter 13), and can quickly correct the requirements.</p> <p>Agile development works well in areas such as web software applications, where a small team can meet the client regularly, and few other stakeholders are involved.</p>	<p><i>Tests are no substitute for Specifications.</i> (Bertrand Meyer)</p> <p><i>Testing can show the presence of bugs, but never their absence.</i> (Edsger Dijkstra)</p> <p>Tests are small samples, which can only cover a small part of a complex system’s behaviour</p> <p>When tests are chosen manually, they are nonrandom samples. Unexpected or extreme behaviours – likely places for trouble – are often not tested.</p> <p>Focusing on testable features may create a bias against behaviours that are harder to test. Qualities (Chapter 6) such as reliability, safety and security are difficult to demonstrate by test. A variety of verification methods may be needed.</p> <p>Specifications are essential to control development of large systems with many subcontractors.</p>

When performances prove to be hard to discover, prototyping (see Chapter 13) is often the best approach. Simulation may also be valuable.

If you cannot identify an acceptance criterion for a function, the requirement most likely needs to be rewritten:

Poor requirement: *The driver must be able to select a radio channel easily.*

Acceptance criterion: ?

You will then need to go back to the source (say, an interviewee) and work out what was intended by the requirement, for example by writing a scenario. Acceptance criteria, like tests, are unforgiving of vagueness, as von Neumann implies in the quote that starts this chapter.

Scenarios/Use Cases

A use case (Chapter 5) organises a group of functions into connected sequences of steps. Acceptance criteria are not usually written for each functional step in

a use case. Instead, postconditions known as ‘guarantees’ are written for the entire structure.

The organisation of functions and performance at different levels is discussed further in Chapter 15.

Success Guarantees

A success guarantee is a condition that must be true at the end of a use case for it to have succeeded. Effectively, this means it covers all the steps leading to success, so it replaces the acceptance criteria for all the individual steps.

For example, at the end of the telematics system use case, ‘Tune the Radio’, the following success guarantees should hold true:

The radio is tuned to the selected channel.

The channel is playing to the selected audio outlets.

Minimal Guarantees

A minimal guarantee is a condition that must be true at the end of a use case, whether the use case ends with success or not. As Cockburn (2001) [7] observes, this is often simpler, more general and easier to keep up to date than trying to identify specific guarantees for each type of failure.

If the use case ‘Tune the Radio’ fails for any reason, the radio promises not to mislead the user about the tuning:

1. *When a channel is tuned, the channel display shows its identifier.*
2. *When no channel is found, the channel display shows ‘No Channel’.*

You may be able to sketch postconditions in a scenario workshop, but by their nature they need careful consideration ‘in slow time’. It is therefore probably better, once you have a good understanding of the wanted scenarios from a workshop, for someone skilled in analysis or testing to draft the postconditions. You should then have them reviewed by the project team.

Performance of a Use Case

It is often more natural to define a required performance for a whole use case than for a single functional step. Recall that a use case describes an episode of behaviour. Stakeholders can relate to the performance of the task as a whole, whereas it may be difficult for people to say how long each step should take. For example:

(Use Case Goal) Specify a Destination

Precondition: User is in ‘Navigation’.

Trigger: User selects ‘Navigate to ...’.

Main scenario:

Telematics system displays menu of ways of specifying destination.

User selects ...

...

Performance: Completed within two minutes for 95% of destinations within the mapped area.

Once a performance value has been set for a whole use case, that value can act as a target. Designers can share out or budget the available time for a transaction among its scenario steps. They can then design each step to meet its derived (budgeted) performance target.

9.2.2 Acceptance Criteria for Qualities

The challenge with discovering what to measure for each quality is that, in stakeholders' minds, qualities are strongly desired but not quantified. Their goals are for products to be safe, reliable, secure and so on. Engineers know that these things come with probabilities and percentages and mean times to failure. Other stakeholders often do not: politicians, for example, feel obliged to say absurd things like: 'Absolute safety is our first priority'.

As stated in Chapter 3, large, top-level quality goals like 'be reliable' are not verifiable. Splitting those goals into smaller, more measurable ones, and then creating workable acceptance criteria for the resulting quality requirements, is the basic discovery process.

There are, therefore, two vital attributes of a good acceptance criterion, and hence of a good quality requirement:

1. It can be measured by a practical and affordable acceptance procedure.
2. Achieving the measured value delivers the quality that the stakeholders asked for.

If the original quality goal has been split into many requirements, an argument through traceability back to the goal is needed to demonstrate that the set of (derived) requirements with their acceptance criteria satisfies the goal (see Chapter 7).

Examples of different qualities are given in Chapter 6. Let us, therefore, just briefly discuss a few which deserve special mention: dependability, security and usability.

Dependability

The challenge in setting measurable dependability targets is twofold:

- it is difficult to make dependability testable;
- it is hard to know how large a value to ask for.

For example:

The telematics system shall have an MTBF of 20,000 hours.

If you drive the car for three or four hours a day, this MTBF (Mean Time Between Failures) equates to between 12 and 16 years of use. Many consumer products, including cars, are discarded long before they are 10 years old, for reasons such as obsolescence, wear and tear, fashion and accidental damage.

Now consider how this MTBF can be verified. Testing for two or more years is generally impracticable, so analysis is the likely route. The analyst will calculate the probability of failure of the whole product from the data sheets supplied by the component manufacturers. Many of the values will be uncertain. Further uncertainties arise from newly-created components, and from software that usually contains unknown design errors (bugs). The real MTBF may be a long way from the analysed value. To be on the safe side, the product is over-designed. This may give it an actual MTBF of 50,000 hours, implying an improbable lifetime of 35–45 years for a family car. In contrast, for intensively used commercial vehicles such as trucks and taxis, such a large MTBF may be essential.

In practice, dependability targets are set very conservatively, especially where safety is involved, as in the automotive domain. Designers of embedded systems therefore try to stay with design approaches that they know are reliable. This conservatism is opposed by market pressure to miniaturise and to move functions into software.

Security

You can't calculate the probability that a system is secure based on the risks it handles, if it's certain that insecure humans will form part of it.

Howard Chivers

The main difficulty in measuring security requirements is that you can neither require absolute security against a given threat (because it can't be guaranteed), nor, in all honesty, specify a probability that a threat will be defeated.

You can specify actions to be taken and defences to be put in place for the purpose of security, but you can't say exactly how much security those steps (that security policy) will buy you. In effect, you have to choose what design steps you feel will be sufficient but affordable: i.e., this too is a trade-off, and you can't avoid responsibility for it (see Chapter 14). With that understanding, security requirements can be specified and measured like any other design constraints.

For example, you can't guarantee that:

All viruses will be caught before uploading to the telematics system.

–a desirable but unattainable goal, if file transfer is allowed at all.

You can specify that:

All incoming files shall be checked for viruses by a commercial antivirus tool.

–a design choice, with risks attached.

Usability

Like performance, usability is strongly affected by the variability of people, their perceptions and experience.

A key to discovering good acceptance criteria for usability requirements is to aim to minimise subjectivity, without making tests too expensive (see 'Usability' under Section 6.4.3 in Chapter 6).

Acceptance criteria can be found for many usability requirements by:

- setting a time target for people to complete a task, using the product;
- setting a statistical target ('95% of people ...');
- specifying compliance with a standard.

For example:

95% of users are able to tune the radio to the desired channel within one minute.

Notice again how closely function, performance and usability are tied together.

Usability analysis and testing are specialist activities [4]. If it is practical on your project, get the help of a usability expert. The discipline may be included in human factors or ergonomics in your organisation.

9.2.3 Acceptance Criteria for Constraints

Constraints are, by their nature, very often quantitative. The acceptance criteria are, therefore, simply that the system meets the required values:

Goal: a satisfying audio experience.

Constraint: The telematics sound system can provide a root mean square (RMS) power output of 10 Watts per channel.

Acceptance criterion: Maximum RMS power output \geq 10 Watts per channel.

Constraints may need to be specified using tables or graphs:

Tele-164: The telematics sound system meets the audio distortion limits shown in table Tele-164a. Allowed distortion at intermediate power outputs (if these are measured) should be by linear interpolation.

Table Tele-164a: Permissible audio distortion curve

<i>Power output per channel</i>	<i>Maximum permitted audio distortion</i>
1 Watt	1%
5 Watts	3%
10 Watts	9%

Acceptance criterion: Maximum audio distortion is within the limits shown in table Tele-164a.

More examples of constraints are given in Chapter 6.

The main challenge is in arriving at a satisfactory value for each constrained quantity, by trade-off between what stakeholders would like and what can realistically be attained. Trade-offs are discussed in Chapter 14.

9.2.4 Verification Method

To make requirements verifiable, you typically need to specify not only acceptance criteria but also the verification method.

The traditional verification methods are:

- test;
- certificate;
- demonstration;
- inspection;
- analysis and simulation (these are often considered the same).

Verification method, like acceptance criteria, can be treated as an attribute of a requirement or of a use case. Table 9.2 shows how one of the constraints just discussed could be handled in a requirements database tool (see Appendix C).

What is striking about this table is the way that it shows the step-wise development of the requirement. The goal effectively states the business/stakeholder need. The justification reflects the engineering challenge

Table 9.2: Verification method and acceptance criteria as requirement attributes.

ID	Goal	Requirement	Verification method	Acceptance criteria	Justification
Tel-174	A satisfying audio experience inside the car.	The telematics sound system can provide a root mean square (RMS) power output of 10 Watts per channel.	Audio laboratory test	Maximum RMS power output \geq 10 Watts per channel.	This is sufficient to provide an intense audio experience despite road and engine noise.

and trade-offs involved in choosing a suitable design to meet the need. The acceptance criterion is the engineering team's response to the challenge. The requirement restates the criterion for contractual purposes.

Let us look briefly at what each of these verification methods means.

Test

*If you don't know what result you expect,
you're not testing, you're experimenting.*

Edsger Dijkstra

A test is a controlled, repeatable procedure with an expected (desired) result. When the verification method is 'test', the acceptance criteria specify the expected test results.

The test method can be subtyped, if necessary. For example, you might distinguish different types of test, such as bench test, field test or stress test.

Certificate of Testing

Testing of non-software products is expensive and also often destructive: testing a product for fire resistance, for example.

In practice, therefore, tests are carried out on a sample. The manufacturer conducts tests under controlled conditions, or pays a standards body to conduct the tests. The product is certified to comply with standards. The results are then accepted by purchasers of the product:

- Consumer goods are typically marked to show standards or regulatory compliance. For example, consumer products that comply with European Union regulations are marked 'CE' on their packaging or base.
- Components for industrial use are typically provided with a manufacturer's data sheet showing what tests have been conducted. These are called 'type tests'. Provided all the components of a given type are manufactured the same way, it is assumed that the sample test results are representative.

Demonstration

Sometimes, a straightforward demonstration that a result can be achieved – a system works under normal conditions – is sufficient verification. Or, a client

may request an informal demonstration under field conditions, as well as formal testing under controlled conditions.

Analysis, Simulation

When a test is prohibitively expensive or impossible, an alternative verification method is to carry out a (mathematical) engineering analysis or, equivalently, a computer simulation, of a product's behaviour.

A Trouble-Free Launch

Large systems often fail embarrassingly in the full glare of publicity on their first operational use, or their first period of especially intensive use, despite extensive and costly 'system' testing.

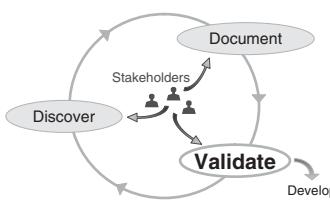
This is because so-called 'system' tests look at **products in isolation**, whereas operations test **products as part of larger systems** involving both people and interfaces (see Chapter 4). For example, an airport's baggage handling system is composed of (at least) the baggage handling machinery and the baggage handling staff, and they have to work together.

Don't even think of saving time by dropping user acceptance testing 'because system testing is so thorough'. That's like saying you needn't run through the end-to-end arrangements for your wedding because you've chosen a good hotel to host your reception.

There is no magic bullet to ensure a trouble-free launch, but some proven practices certainly help:

- **Go-live readiness review:** are all the preconditions for correct working in place? Major systems have fallen over because nobody thought to ensure that all their operators had working usernames and passwords on day one, or cards for the company car-park. Attention to detail is everything here.
- **Dress rehearsal (launch scenario):** play through the whole launch exactly as it should go on the day, with the actual staff, etc. Don't leave anything out.
- **Incremental commissioning:** can you launch your system or service in easy stages, correcting any small problems as you go along? It's much safer.

9.3 Validating Acceptance Criteria



The proof of a pudding is in the eating.
English proverb

The proof that a requirement is well written, and especially that acceptance criteria are adequate, is when a good product emerges.

Completeness of testing (or other forms of verification) has several components:

- Every requirement has a verification method and acceptance criteria (i.e. it has a defined and realistic test).
- Every scenario path defined in a use case is covered by a planned test.
- Sufficient other tests are provided to cover important practical situations adequately, e.g. specific end-to-end scenarios in specific combinations of environmental conditions, load, etc. As 100% coverage is impossible here, professional judgement is required.

For example, any handheld device needs to tolerate a range of temperatures, humidity and shocks. These environmental factors may be more damaging in combination, e.g. hot, moist, tropical conditions; or being dropped onto hard ice in extreme cold. Where these are likely scenarios of use, combined tests should be specified.

Obviously, if testing does not begin until the end of development, that is too late to find out if either your requirements or your tests were good.

9.3.1 Testing from Day One

Fortunately, you can start looking at the testability of your requirements very early in development:

- In an iterative or agile life cycle, testing begins early. This inevitably exposes poor acceptance criteria.
- Tests and other forms of verification can often begin early, using models, simulators or special-to-purpose test equipment.
- Experienced test engineers can quickly explore possible tests of requirements in their minds, and propose effective acceptance criteria. Therefore,

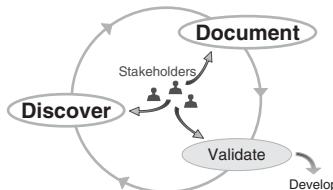
as already suggested, you should get test people involved as early as possible.

- The project team can ‘walk through’ scenarios (or use cases) as if they were running them as tests, and can check that the postconditions (guarantees) are what they would expect to see as desired results.

Tips for Acceptance Criteria

- Consider how you will test (or otherwise verify) each requirement.
- If you find a requirement is too vague to verify (e.g. ‘typically’, ‘generally’, ‘easy to use’, ‘responsive’, ‘efficient’), mark it as unsatisfactory and work on it some more.
- If a requirement is too sweeping to verify (e.g. ‘always’, ‘never’, ‘under all circumstances’, ‘for all users’), then again mark it as unsatisfactory and work on it some more.
- Get your test engineers involved – they are expert at spotting weak requirements.
- You may need several tests, and hence several acceptance criteria, for some requirements.
- If special test equipment (e.g. network simulators, test harness) is needed, make sure it gets included in the requirements and project plans.
- If testing is not possible, work out what else (simulation, formal proof, etc) you may need, and plan for it early.

9.4 Measuring Quality of Service (QoS)



To measure is to know.

William Thomson (1st Baron Kelvin)

The reason a business writes requirements for systems and products is to obtain the results that they can provide. But specifying a product is only one way to obtain a desired result.

An alternative route is to write requirements for a service. This may cause a service provider to operate the same products as you would have obtained for yourself. The approach is clearly less prescriptive: you are specifying the results, but not the means of delivering them. This has both advantages and disadvantages:

- Advantage: the service provider can find a more economical way of running the service, which may bring benefits to both client and provider. You are not insisting on products which may quickly become obsolete.
- Disadvantage: the service provider can analyse the cheapest way of providing the results, possibly without creating the products you expected (and perhaps hoped for).

9.4.1 Example Service: Office Carpeting

Let us consider an example, to see how meeting a need with a service compares with meeting a need with a product. There are both similarities and differences.

If you want to have something quiet, comfortable and warm to walk on in your office, you could go out and buy a Persian carpet which has these qualities. It will be just the kind of carpet that you like, but if you spill something on it, you are responsible for cleaning or replacing it. You carry the risk directly.

In contrast, an office carpeting contract may tile your office floor with squares of tough industrial carpet. These may not look as good as your new Persian carpet, but if you spill something on a carpet tile, the carpeting provider will replace the dirty tile at low cost and effort. The contractor carries at least some of the risk.¹

The requirements for the carpeting contract are shown in Table 9.3.

The quality of the carpeting service is measured by how well these requirements are met. However, they differ in how they can be measured:

- Requirements 1 and 2 would be the same whether the carpet was bought, made or provided as a service. They can be verified at acceptance time, by inspection and certificate of testing.
- Requirement 3 applies to the continuous provision of a service, and should be measured while the service is being provided. ‘QoS measurement’ is therefore added to the list of types of verification method. The ability of the service to carry out the task (once) can also be tested (before acceptance)

¹Major business risks cannot be removed so easily. Suppose your business critically depends on its IT network. An IT service provider can run your IT network for you, but if the network fails entirely and the IT provider fails to support you as agreed, or the IT provider goes bankrupt, your business will be disrupted whether or not you later manage to recover payments in compensation.

Table 9.3: Requirements for a service.

ID	Requirement	Verification method
1	The floor of the office must be covered with carpet.	Inspection
2	The carpet must not slip when walked on.	Certificate of testing
3	Any dirty or damaged carpet must be replaced with new carpet within 48 hours of being reported.	Test; QoS measurement

by deliberately damaging some pieces of carpet, so ‘test’ is also a valid but not in itself sufficient verification method.

Possible QoS Measurements

Someone in your organisation will have the duty to measure compliance with requirement 3 in Table 9.3, and will keep statistics like: ‘In the second quarter of this year, 97% of carpeting requests were met within the time limit’. Possible quality of service (QoS) measures for requirement 3 include:

- percentage of requests not met within time limit (per month or quarter);
- number of requests not met within time limit (per month or quarter);
- average time to meet a request (per month or quarter).

These can give very different results; payment for carpeting may depend strongly on the measure you choose.

Suppose there are 10 requests not met on time in a particular quarter. If the quarter is a busy time for requests, this may be a small percentage of the total number of requests, so measure (a) is small. On measure (b), ten may be an unusually large score.

Now, suppose that in another period, there is just one late request, but an extremely late one, out of an average number of requests. On measures (a) and (b), the score is low. On measure (c), the score is high.

QoS is notoriously subjective. Perceived service depends on people’s expectations and their emotional state (which is coloured by other things they experience at the same time). It is difficult to measure a service exactly at the point that people experience it. And a statistical analysis that says that 93% of the trains were on time last year does not necessarily improve the feelings of the people who were on the 7% that were late. See Rob Eastaway’s (2005) book, *Why Do Buses Come in Threes? The Hidden Mathematics of Everyday Life* [6].

9.4.2 Two Opposite Approaches

Almost any desired result, whether for a business or for personal consumption, can be delivered as a service.

For example, a hospital has a choice in how it organises its cleaning services:

- **In-house development approach:** the hospital specifies its cleaning regime, and works out what staff and equipment it needs to meet that regime. The hospital buys that cleaning equipment and hires the cleaning staff. It instructs its cleaning staff to follow hospital procedures for cleaning. If the equipment fails, the hospital repairs or replaces it.
- **Service contract approach:** the hospital specifies its quality of service (QoS) requirements for cleaning, and lets a contract to a cleaning company. The hospital continuously monitors the cleanliness of the hospital according to the QoS measures. The cleaning company obtains and maintains whatever staff and equipment it needs to meet the required QoS.

In a hospital, the same choices can apply to laundry, catering, patient telephone and entertainment, car parking and other services. They also apply in most types of business, whether commercial or not.

For example, a business can procure its own computers and communications equipment, and hire IT staff to install and maintain it; or it can pay a service company to provide a so-called ‘turn-key’ IT service (the supplier does everything: you just turn a key and it all works).

9.4.3 A Spectrum of Service Approaches

It is simplistic to consider that the choice of means of delivery is only between development and services, however. Let us consider how services can be used to meet a basic need: food.

In personal life, food can be purchased raw, and prepared and cooked at home; or you can go to a restaurant for a complete service. This traditionally limited set of choices has been greatly extended by modern commercial approaches. Meals are now prepared in several commercial scenarios, which illustrate the range of ways that services can be provided in almost every walk of life (Table 9.4).

Table 9.4 shows that the traditional restaurant lies at the extreme of a spectrum of service provision, where almost every step in the chain of meal delivery becomes a service, and essentially no food-processing products or equipment remain to be purchased by the consumer. The other entries in the table show that intermediate approaches are commercially viable; indeed, you will be able to think of more examples. The same applies to many other industries.

Table 9.4: Scenarios and commercial services for food preparation.

Approach	Services	Scenario
100% home grown (You have to obtain all needed equipment)	None	You grow your own vegetables (using your own garden and gardening tools), pick them, prepare them for cooking (using your own kitchen equipment), cook them (using your own stove), eat them and wash the dishes yourself.
Market	Food production, limited distribution	A farmer grows the food and brings it to market once a week. You buy the food, wash it, prepare it, cook it, eat it and wash the dishes.
Supermarket	Food production, washing, packaging, distribution	A farmer grows the food and sells it to a supermarket company. It washes and packs it, and trucks it to a supermarket in your town. You buy the pack, discard the packaging, prepare and cook the food, eat it and wash up.
Ready meal	Food production, washing, preparation and cooking, packaging, distribution	A farmer grows the food and sells it to a supermarket. It prepares it for cooking, cooks it, and packs it (and optionally freezes it). You buy the pack, discard the packaging, briefly heat the food, eat it and wash up.
Takeaway; home delivery	Food production, distribution, washing, preparation and cooking, packaging, delivery	A farmer grows the food and sells it to a distributor. A business buys the food and prepares it for cooking. You order and pay for the dishes you want. The business cooks and packs the food. You discard the packaging, eat the food, and wash up. Variation: you order by telephone or Internet. The business delivers the food to your front door.
Restaurant (All needed equipment belongs to the restaurant and other businesses)	Food production, distribution, washing, preparation and cooking, packaging, presentation, table service, washing up and cleaning	A farmer grows the food and sells it to a distributor. A restaurant buys the food and prepares it for cooking. You go to the restaurant and sit at an elegant table with a clean tablecloth. You order the dishes you want, in person. The business cooks the food, and serves it to you at the table. You eat the food, pay, and leave. The restaurant does the washing up. A laundry cleans the tablecloth.

Unaffected by Means of Delivery?

The fact that the scenarios in Table 9.4 are all different shows that the means of delivery (developing a product versus paying for a service) affects the design of a system (and the products within it). They also affect the customer.

In theory, behaviour visible on the outside of a system (i.e. at its interfaces with its users and the rest of the world) should be unchanged, whatever the means of delivery. This accords with the dogma that requirements should be free of design and should say only what results are desired, not how they are to be delivered.

In practice, it is difficult to make a service entirely ‘transparent’, i.e. invisible to a system’s users.

For example, users notice that an IT service is not locally provided when something goes wrong, e.g. a desktop computer starts to malfunction. They have to place a service request via someone who does not know their office, and often have to restate the details of their problem to different people (call centre staff, technical advisors, maintenance technicians).

In contrast, a company’s own maintenance technician would know the desk concerned and would soon build and remember a picture of the problem. This may mean that the in-house approach offers a better QoS for company staff, but maybe at higher cost, complexity and risk for the company’s management.

Perhaps a general rule is that the more naturally ‘virtual’ a service is, the less it matters who provides it, or where. The more a service depends on local knowledge, memory of a case, experience, skill, or specific physical resources, the less suitable it is for ‘outsourcing’ to a generalised ‘service provider’.

9.4.4 Worked Example: QoS Measures for Food Preparation Services

The differences between the six food preparation approaches in Table 9.4, as far as can be observed by the consumer, may include:

- standard of food production (organic, free range, etc);
- freshness of the food;
- cleanliness of the food (during storage, distribution and preparation);
- excellence of the cooking;

- presentation of the food;
- reduction of effort (required to prepare, cook, wash up, etc);
- price.

Let us consider how suitable these qualities are as QoS measures. This will lead to some general conclusions about choosing QoS measures for service requirements.

Reduction of effort and **price** form the benefit/cost rationale (see Chapters 7 and 10) for selecting the appropriate approach. Rational consumers are prepared to pay for a service if it reduces the effort they have to make. In the case of a business customer, a service is worth paying for if the benefits to the business are more than its cost. This is the basis of 'value engineering'. However, many factors affecting business decisions cannot properly be evaluated in financial terms (see Chapter 14).

Standard of production, freshness, cleanliness, excellence of cooking and presentation are *specialised qualities* suited to the business of food preparation and delivery (see Chapter 6). All of them are important to the person eating the food. However, they are not necessarily easy to measure.

- **Excellence of cooking** and **presentation** can be measured, subjectively at least, at the dining table. They form the easiest qualities of service (QoS) to measure, so people are likely to choose these.
- **Cleanliness** and **freshness** of food storage and distribution often cannot be observed directly from the dining table, so two choices are available for measuring this QoS:
 - the easy but possibly misleading route is to observe food preparation, but good food preparation does not show that the food was not contaminated earlier in the chain;
 - a better measure of QoS is to inspect each stage of food storage, distribution and preparation.
- **Standard of food production** cannot be observed directly, except by visiting farms. Inspection regimes are provided by organic certification (for organic produce only) and by the regulation of normal farm production.

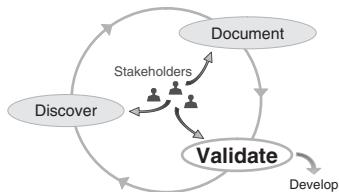
Measurement of food quality is thus quite complex, and it cannot necessarily be done well with measurements made only at the point of delivery. In other words, services cannot always be treated as 'black boxes'. Such complexity is common in QoS measurement.

Subjective measures are in wide use. For example, restaurant reviews published in newspapers are often the opinion of one critic, made on the basis of a single visit. Subjectivity is undesirable in QoS (as in acceptance criteria) because of its potential unfairness.

Services can be assessed more objectively by:

- **avoiding subjective QoS**, e.g. choosing to measure presentation, but not excellence of cooking. Presentation can be measured on a defined measurement scale (e.g. 1 = dirty tablecloth, plates, cutlery or glasses; carelessly placed food; . . . 5 = spotlessly clean tableware, elegant placing of food) to improve the repeatability of assessments, and assessors can be trained and monitored. Note that the measurement scale in this case is qualitative: both quantitative and qualitative measures may be affected by subjectivity;
- **repeated (statistical) sampling**, e.g. visiting a restaurant at least three times to make a balanced judgement (a restaurant might have had a bad day on the first visit), although this takes more time and costs more;
- **intersubjective agreement**, e.g. evaluation of excellence of cooking by a panel of three assessors. This approach enables even ‘subjective’ QoS measures to be scored repeatably but, again, this increases the cost of each measurement.

9.5 Validating QoS Measures



9.5.1 Qualities of a Good QoS Measure

We are now in a position to describe what makes a good QoS measure.

QoS measures should be:

- **repeatable** – the measuring techniques give the same answer each time on a service of the same quality;
- **affordable** – measurements are not too costly to make;
- **objective** – the measurements avoid subjective bias;
- **representative** – the measurements are of a true sample of the service (or, the whole service is measured). In the case of a restaurant evaluation, this may mean secrecy; restaurants do not know which diner is evaluating them for the *Guide Michelin*, for example, so they cannot give the assessor unrepresentatively good service. As another example, ambience on the London Underground is evaluated by ‘mystery shoppers’ who visit

stations and trains to assess cleanliness,² lighting, passenger information and so on;

- **To the point** – the measurements reliably distinguish between good and bad quality services. This means (as with good acceptance criteria) that the QoS force the service provider to deliver the services that the stakeholders actually need.

As with acceptance criteria, completeness is not easy to specify for QoS measures. Clearly, each required quality must be measured, but beyond that it often does not make sense to try to measure everything.

Tips for Choosing QoS Measures

Services need to be measured quasi-continuously. So:

- Choose measures of quality that you can reliably collect, day in, day out.
- Measures of quality, like acceptance criteria, should give a definite answer to the question: 'Is this level of service acceptable?'
- Avoid measurements that are easy to collect (e.g., number of requests for a service) but which do not tell you whether the service is performing well.
- Look at the complete set of QoS measures that you have, and ask yourself: 'If the service did well on all of these, but only these, measures, would we be happy with it?'

9.5.2 Will your QoS Measures Work?

Not all that counts can be counted; Not all that can be counted, counts.

The Metricator's Maxim

It is not easy to specify QoS measures that work reliably for long periods. Service providers will naturally adapt their provision to maximise their profit.

So, for example, if you choose to measure a hospital's performance by how quickly it moves casualties from newly-arrived ambulances to actual

²An urban legend sprang up about this particular QoS measure. It was popularly reported that the assessors had to count the number of pieces of waste paper over 2 cm long on each railway platform. In reality, the criteria looked at whether the platforms had been swept and, if appropriate, polished; whether the walls were free of graffiti, and so on.

treatment, the hospital may arrange to keep its score high on that measure by instructing ambulance crews to drive around the town (with the patients on board) until the casualty department is ready to receive more work. The ‘letter of the law’ is complied with, but the spirit of the measure is broken. QoS measures and targets can thus have undesired effects.

Unfortunately, systems tend to become ‘immune’ to any particular set of tests (often called ‘benchmarks’). This is because people are intelligent and creative, and gradually find ways to work around QoS measures.

For instance, if you were always to take measurements at midday, your supplier could learn to arrange for most service outages to occur between 3 pm and 5 pm, causing you maximal disruption.

The best solution is a mechanism that allows you to inject new tests or measures from time to time, to ensure that the ‘spirit of the law’ is complied with. You might take measurements at random times, for example.

9.5.3 Common QoS Measures

One final question is whether there are typical qualities of service, which you can safely reuse.

A possible answer is simply: ‘No; services, like products, are all different’.

However, some qualities seem to occur in many contexts:

- **availability** of service (can you have it when you want it?);
- **speed/performance** of the service (do you have to wait?);
- **accuracy/correctness** (do they do exactly what you ask?).

It is probably always worth considering how you will measure these three qualities when you are specifying a service.

9.5.4 Validating QoS with Negative Scenarios

You should also consider what side-effects choosing such measures may have, remembering that your service provider will seek to make savings in unmeasured areas.

You might decide to run a negative scenarios workshop (see Chapters 5 and 12) to look for possible weaknesses in your QoS approach.

For instance, you can appoint a ‘red team’ to devise ways of getting around your QoS measures so as to maximise the service provider’s profit. The task of the ‘blue team’ will be to devise QoS measures to force the provider to give the wanted level of service. Of course, the red team gets the chance to devise further tricks, so the workshop has the character of a chess game.

9.5.5 Things To Check Measurements Against

- The requirements being measured (Chapters 4, 5, and 6)
- Targets mentioned in interviews and workshops (Chapters 11 and 12)
- Measurement approaches used on previous projects (Chapter 13)

9.6 The Bare Minimum of Measurement

- Make sure you'd be happy if you *only* got *exactly* what you're asking for.
- But first, make sure your goals are right (Chapter 3). There's no point specifying precise measurements of the wrong deliverables.

9.7 Next Steps

Prioritise your requirements (Chapter 10).

9.8 Exercise

Rationale analysis suggested some possible new requirements and measurements for the multi-function device for outdoor leisure use:

- a. The device shall weigh no more than (TBD 800) grams.
- b. The device shall be operable between (TBD -50) and (TBD +60) degrees Celsius.
- c. The device shall be waterproof to (TBD 3) atmospheres pressure.
- d. The device shall survive a drop of (TBD 1) metre onto a hard surface.
- e. The device shall be rechargeable in the field in (TBD 30) minutes.

For each TBD measurement, state the factors to be traded off when considering what value to choose, and identify how you would decide on the right value.

Hint: think about what issues are *essential* to device users.

9.9 Further Reading

1. Alexander, I. and Stevens, R. (2002) *Writing Better Requirements*, London: Addison-Wesley.
Provides a simple description of how to frame individual requirements so they can be tested easily.

2. Robertson, S. and Robertson, J. (2006) *Mastering the Requirements Process*, 2nd Edition, Upper Saddle River, NJ: Addison-Wesley.
Gives a good straightforward account of 'fit criteria' (acceptance criteria).
3. Cohn, M. (2004) *User Stories Applied: For Agile Software Development*, Boston: Addison-Wesley.
This is a clear description of the philosophy and practice of agile development, including how it makes use of scenarios and acceptance tests.
4. Kuniavsky, M. (2003) *Observing the User Experience: A Practitioner's Guide to User Research*, San Francisco: Morgan Kaufmann.
A readable and practical guide to usability testing.
5. Lauesen, S (2007) *Guide to Requirements SL-07: Template with Examples*, Copenhagen: Lauesen Publishing.
A template with useful examples of software requirements.
6. Eastaway, R. and Wyndham, J. (2005) *Why Do Buses Come in Threes? The Hidden Mathematics of Everyday Life*, London: Robson Books.
An entertaining book that deals with the relationship between system behaviour and the perceived results for users – such as why a regular schedule of buses always tends to translate into clumps of activity with long gaps in between.
7. Cockburn, A. (2001) *Writing Effective Use Cases*, Boston: Addison-Wesley.
Cockburn proposed the use of success guarantees and minimal guarantees as measurements for use cases.

CHAPTER TEN

Priorities

When negotiating requirements, it is better in the early stages to replace words that have nonnegotiable connotations such as 'requirements' (things claimed by right and authority) with words like 'objectives' and 'goals'.

Barry Boehm

Requirement Elements	Priorities
Discovery Contexts	
Introduction	
From Individuals	
From Groups	
From Things	
Trade-Offs	
Putting it all Together	
Stakeholders	

Answering the questions:

- Which requirements shall we decide to meet, given we can't do all of them?
- What is triage, and how should we do it?
 - ... so you don't waste time on things people don't need
 - ... so you get the best value given a limited timescale and budget.

10.1 Summary

Discovered requirements may differ in their value to stakeholders; and they may not be practical and cost-effective in the context of the chosen design. Prioritisation, therefore has two key aspects: looking back to what stakeholders want (input); and looking forward to what can be done (output).

Input prioritisation identifies which goals (or groups of goals related by dependency) stakeholders want most. The purpose of this is initial triage, in which you reject (or defer) the goals that are not worth working on. The result is a list of goals that defines the purpose or value of the product or service to the stakeholders. You also have the opportunity to capture stakeholders' rationale for wanting or not wanting each goal.

Output prioritisation considers whether the goals can be achieved in the chosen design at acceptable cost and risk. This entails optioneering, i.e. identifying the possible design options, and evaluating which option best meets the goals. That in turn enables the cost and risk of each requirement to be identified. A second triage step may then reject the least implementable requirements. You have the opportunity to capture the design rationale, which may be useful if decisions are ever challenged, and on later phases and projects.

The result is a list of requirements that are both wanted and known to be implementable, and at the same time a design approach that is known to be able to implement the requirements.

10.2 Two Kinds of Priority

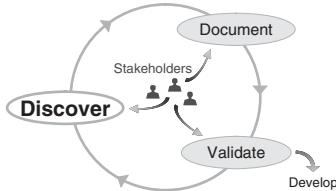
The crucial thing to understand about prioritising requirements is that the task has an input side, from stakeholders, and an output side, with respect to design:

- **input priority** – what the stakeholders most want (realisable or not);
- **output priority** – what can be realised in a practical design.

Let us take each of these through the discover, document and validate stages of the requirements cycle in turn.

10.3 Input Priority

10.3.1 Discovering Input Priority



On the input side, you have stakeholders' goals. You can try to find out which goals are most important to them. The task demands some skill, as the most likely answer to naive questions like:

'Which of these goals do you most want?'
or
'Which of these are absolutely essential to you?'

is:

'All of them'.

Still, it is possible to get people to compare goals, for example in a stakeholder workshop (see Chapter 12) or if your team assesses priority independently. The discussion and arguments for and against different goals can be captured as part of the project's rationale (see Chapter 7).

Triage on the Input side, based on Value to Stakeholders

The purpose of discovering how important each goal is to stakeholders is to decide whether it should go into the product. That decision may be taken at any time during development:

- At the earliest, you may see that a goal is not worth pursuing, even as far as getting stakeholders to prioritise it.
- Analysis of trade-offs of goals against design options may show that some goals cannot be met in the chosen design, setting the output priority (see Section 10.4).
- At the latest, if testing reveals severe problems with a feature, that feature might be dropped just before the product is released.

The accept/reject process is always somewhat brutal, and we have seen requirements managers who are slightly ashamed of how they do it. It is called triage.

Triage¹ means sorting candidate requirements – goals – into groups. In its original medical usage, it meant looking at casualties to decide how to use the scarce resources available to treat them as effectively as possible. Casualties were sorted into:

- trivially injured, not needing immediate treatment;
- untreatable, not able to benefit from treatment;
- seriously injured, likely to survive if treated immediately.

In requirements work, you can have two, three or more triage groups as convenient to your project. You can apply triage on both the input and the output side. We describe triage on the output side in Section 10.4.

On the input side, triage principally means filtering a list of proposed goals to discard any that are certainly not worth proceeding with. The basic way to do this is:

1. Obtain stakeholders' priorities.
2. Set an accept/reject threshold.
3. Discard all goals that have priorities below the threshold.

A slight variation is to set two thresholds, creating three categories: accepted; deferred (saved for later); and rejected.

You cannot tell, purely from stakeholder (input) priorities, whether a requirement is *practical*. Questions of how long it will take to implement a requirement, how much it will cost, how risky it is and hence whether it is worthwhile, can only be answered by considering impact on design, which means triage based on output priorities.

Techniques for obtaining stakeholder priorities include:

- independent experts;
- voting, valuing, consensus, panel decision, etc;
- card sorting;
- survey, sample, etc;
- matrix techniques.

These techniques are described below.

In addition, product managers may derive goals from problem reports or customer suggestions. You (on behalf of your stakeholders) should filter these to obtain a list of changes that are worth making to your product. Chapter 13 illustrates a process for doing this.

¹The word triage comes from Old French *trier*, meaning to pick or sort, related to the English words *try* and *trial*. Its modern meaning seems to have been influenced by the unrelated Latin word *tria*, three.

Techniques for Discovering Input Priorities

Many techniques are possible for discovering input priorities. We will mention just a few. For now, we will assume that the goals being prioritised are all independent; we will explore what to do when some goals depend on others later in this chapter.

The Law

Some requirements are imposed by law, for example, safety standards for electrical equipment. Requirements of this kind are typically not negotiable, so they appear as fixed, externally imposed constraints on your project (see Chapter 6). Their input priority will be ‘mandatory’. You should take care not to give this maximum priority to other requirements that are not quite so firmly fixed.

Independent Experts

Requirements may be discovered by different people – the security expert does the security chapter, and so on. In that case, the rest of the team may have little ability to review or prioritise the proposed requirements. Your best bet is probably to have the security requirements checked, and perhaps prioritised, by an independent security specialist, and so on for each kind of requirement.

Voting, Valuing, Consensus, Panel Decision, etc

If you are working on a group of requirements (e.g. product functions) as a team, then workshop techniques are appropriate. A facilitator can present each requirement; the team can simply vote, negotiate for consensus or have a chairperson, such as a team leader, use a casting vote to identify input priority. Or, a panel can be set up for the purpose.

Al Davis (2005) suggests some enjoyable and practical team prioritisation activities in his book *Just Enough Requirements Management* [1]. For instance, you can arrange the requirements on the walls of a workshop room, and give participants \$100 each in toy money. The task for the participants is to allocate their money to the requirements, however they choose. The requirements with the most money are accepted (Figure 10.1).

Techniques like this need to be used with care. For example, requirements affecting security (say, the switches a system administrator can set to disable specific features) might be considered of little interest by many stakeholders, but may still be critical. On security matters, it may not be right for the security person to have the same voting power as everyone else. The same may apply in other areas such as safety, reliability and so on.

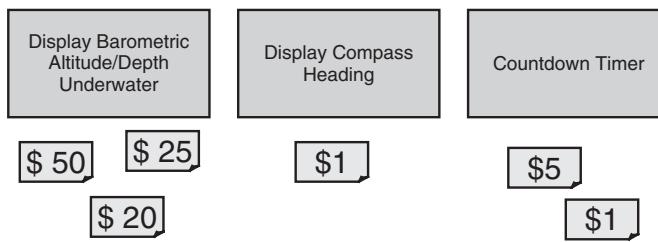


Figure 10.1: Voting by allocating toy money.

Sorting out Priorities from Software Problem Reports

Software products usually go through many versions, and many people may have trouble using them when they first start. This means that users send in problem reports for a wide variety of issues.

- Some will be serious, indicating real errors in the software, or things that the software really ought to do. Those should lead to corrections (bug fixes) or urgent new requirements that will make your product better – more useful, more reliable and easier to operate.
- Others will be minor, for example indicating small inconsistencies in formatting or style, or small glitches such as when a window fails to refresh straight away. Those should lead to non-urgent corrections or requirements. Crashes triggered by rare and unlikely combinations of user inputs may also be given low priority.
- Finally, you will get many reports of ‘software problems’ that you can’t quite understand or, even if they seem clear enough, you can’t observe, no matter how hard you try to get them to happen on your copy of the software. (Software support people call this ‘failing to replicate the problem’.) The user may have been running the software on a network that was overloaded, or on a computer that was running out of memory, or at the same time as some other software that momentarily interfered with it. Or perhaps the user was confused by something out of your control, and attributed it to your software. Whatever the reason, you need to triage out such reports. That may mean gently asking the user concerned to provide more detailed evidence; certainly, if the problem recurs it must have a cause.

Card Sorting

If the requirements come from a small number of experts, you can ask each expert to sort a deck of cards, each with one requirement written on it, into an

order from the most needed to the least needed. This directly yields a ranking (first, second, third, etc).

A variation (used in the Analytic Hierarchical Process (AHP) described in Chapter 14) is to ask the experts to compare requirements pairwise. An ordering can then be calculated, and any inconsistencies (e.g. A > B, B > C, but C > A) can be explored by discussion.

Similar cautions apply as for voting. It only makes sense to compare the need for things that are optional, and independent of each other.

Survey, Sample, etc

If the requirements are for a mass-market product or service, you can use market survey techniques to estimate which requirements are most popular.

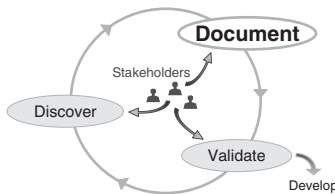
Or, you can invite in a panel of consumers and find out their preferences. That can be by facilitating a discussion and observing their stated opinions, or by a technique such as card sorting.

Again, this approach does not identify what may be critical to the working of the product or service – there may be dull infrastructural requirements that stakeholders do not think about, but which are absolutely necessary.

Matrix Techniques

Techniques such as QFD that compare proposed elements (e.g. features) on various criteria (e.g. cost) are discussed in Chapter 14, in the context of evaluating design options against goals. Such matrix techniques can equally be applied to evaluating product goals against project or business criteria, to answer the basic prioritisation question: which of these things do we most want to do?

10.3.2 Documenting Input Priority



The input priority is the result of merging the beliefs of all the stakeholders on the importance of each requirement, feature or scenario. It can be recorded on any convenient scale, such as {Mandatory, Desirable, Possibly Useful, Luxury}, and held as a requirement attribute (Figure 10.2).

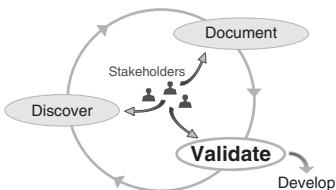
ID	Function	Priority to Stakeholders
Func-321	Print an address label.	Choose a value: ▽
		Mandatory
		Desirable
		Possibly Useful
		Luxury

Figure 10.2: Documenting input priority.

Other input priority scales in common use include:

- {9 – Key, 3 – High, 2 – Medium, 1 – Low}
 - The use of integers on their own introduces ambiguity – is 1 higher or lower than 3? – so it is always best to use an explanatory word as well.
 - ‘Key’ is meant to indicate one of a very few (say, you are only allowed seven) absolutely critical requirements, i.e. a design will only be accepted if it satisfies this requirement. It means a higher priority than ‘High’. The idea is to force stakeholders not to give all their requirements the highest possible input priority.
- {Must have, Should have, Could have, Won’t have}
 - This is popularly remembered by the acronym MoSCoW. This does not constitute a prioritisation method, just a hint that stakeholders should somehow assign and record priority at these four levels. Despite being essentially an input priority (what we want), the MoSCoW phrases feel uncomfortably non-negotiable. They make most sense with an iterative life cycle: see the ‘Cherry Stones’ approach in Section 10.4.2.

10.3.3 Validating Input Priority



Prioritisation is a process with many stages, offering numerous opportunities, both formal and informal, to check that priorities are correctly assigned, and indeed that requirements are valid and realistic.

Ways that you can validate priorities include:

- **walkthrough with stakeholders** (including designers):

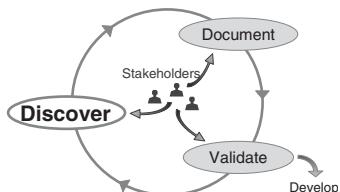
- a simple workshop to step through the requirements and ensure that the assigned priorities make sense, e.g. ‘... and we’ve agreed that the iTunes player function is deferred to Phase 1b?’;
- **deciding how many requirements really are top priority** ('essential', 'mandatory', etc):
 - if everything is top priority, it will be difficult to identify which requirements to defer if the project overruns. Use a mechanism (such as voting, card sorting, etc) to identify requirements that could safely be deferred;
- **checking priorities against scenario and infrastructure dependencies:**
 - identifying dependencies can reveal inconsistencies in priority. In particular, if two requirements are tied together, e.g. by being needed in the same scenario, their priorities should match. Dependency is discussed in Section 10.4, Output Priority.

In principle, you should not assign a priority to a requirement until it is otherwise completely and correctly documented, so prioritisation is an opportunity to validate all requirement attributes and traces. For example:

- a requirement that does not trace to any goal may offer no benefits to stakeholders – it should either be justified or deleted;²
- a functional requirement without traces to scenarios cannot be checked properly for possible dependencies.

10.4 Output Priority

10.4.1 Discovering Output Priority



On the output side, you have to discover what your project *should* work on. Starting from the input priority list, your task is to decide which requirements you can actually afford to include and can risk including.

²This is popularly known as a 'gold-plating' requirement: it may seem attractive, but adds no improvement in functionality to the product.

But how can you tell, just from a list of requirements, which are affordable and safely workable? Clearly the answer must be 'by working out what each requirement implies for the design of the system'. That could mean:

- having a skilled designer quickly think through a new software function's likely complexity, risk, and size in lines of code. This may take a few minutes or a few days;
- preparing several candidate designs for a system, and evaluating both their cost and how well each of them meets the requirements. This may take many months.

The first of these processes is often covered by vague concepts like 'costing the requirements'. The second is called trade-off analysis or 'optioneering' (see Chapter 14).

The two processes are in essence the same, even if they look and feel very different: the requirements and the design are developed until they are sufficiently well understood, and then traded-off. The result in both cases is simultaneously:

- an agreed list of requirements that will be developed;
- a design that is known to be a workable way of meeting the agreed requirements;
- a rationale for the decisions made.

There is an enormous variety of scale and complexity among projects. On a small project, output prioritisation may be quite informal, involving no more than a discussion over a whiteboard or a spreadsheet. On a large project, prioritisation may be a process of many stages, involving carefully separated stakeholder requirements, system requirements and subsystem requirements, with traceability between them. The discussion that follows ignores these differences to focus on the essential concepts.

Triage on the Output Side, based on Knowledge of the Design

If the chosen design cannot meet a given requirement, that requirement cannot be implemented. Conversely, if a requirement is actually critical – the product will be useless if the requirement is not met – then any design that does not meet that requirement must be rejected. In this way, both candidate requirements and candidate designs are subject to triage. Notice that this is a *second* application of triage, entirely separate from triage of requirements based on input priority.

Therefore, setting an output priority is not a clerical matter, merely quickly filling in a database attribute. Discovering the value of such an attribute may

involve months of analysis and design, depending on the type of system your project is developing. You will have to consider both:

- the demand side – the benefit to stakeholders and the value to your organisation; and
- the supply side – the cost and risk of the development, based on optioneering (see Chapter 14).

Risk and cost both depend critically on how well your team knows how to develop the required kind of system. Innovation is inherently risky. The more similar a new product is to one you have developed already, the lower the risk.

Life History of a Requirement

We are now in a position to sketch the life history of a typical requirement's discovery (shown as a UML state chart in Figure 10.3) and, in a way, to summarise the message of this chapter.

Requirements essentially have an uncertain status until they reach the optioneering decision point. While they are being negotiated, as Barry Boehm suggests in the quote that introduces this chapter, it is probably best to speak about them as 'goals'. After that point, they certainly deserve to be called requirements. But industry is used to calling all these things 'requirements', and we can't avoid it.

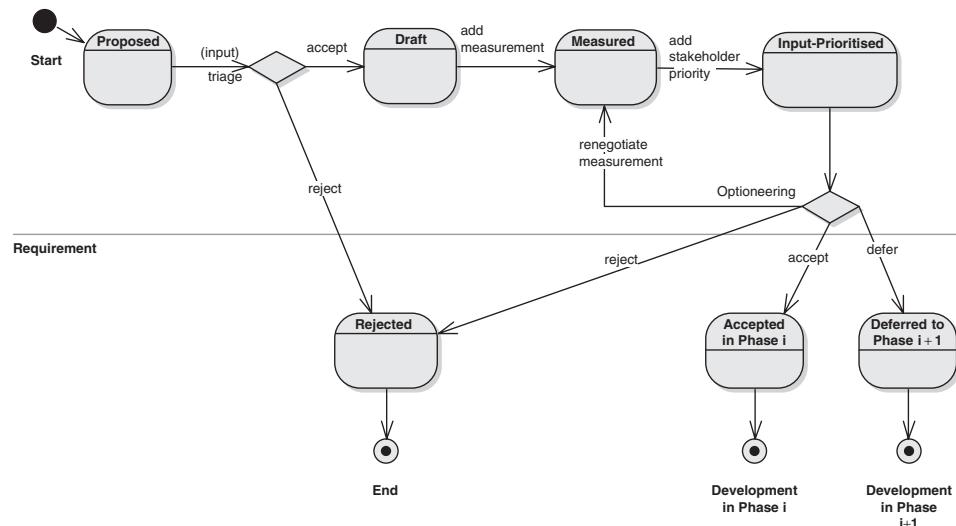


Figure 10.3: State transition diagram of requirements through triage, prioritisation and optioneering.

Figure 10.3 also suggests that requirements can be developed in different programme phases. A programme is a sort of ‘superproject’ composed of a sequence of project-like phases (and perhaps other projects in parallel). Clearly, there could be any number of these. Note that if a requirement is deferred to a future phase, it may well have to undergo a further round of optioneering. More is said about how requirements relate to programme phases in Section 10.4.2, Documenting the Output Priority.

On a large project, the ‘goals’ of the upper half of Figure 10.3 will be recorded in a stakeholder requirements document, while the ‘requirements’ of the lower half of the figure will be recorded in a system requirements document (i.e. treated as separate entities from input-prioritised goals). Further implied ‘system’ requirements are often added at this stage, for example to ensure compliance with standards.

A Worked Example

Let us follow a requirement through the stages shown in Figure 10.3. Suppose a goal for a multifunction device (MFD) is to help walkers and climbers predict changes in the weather. (See Figure 10.4 for a scenario.)

That could be realised through subgoals such as providing a barometer function (displaying atmospheric pressure). We may note in passing that a barometer (a design element) can also provide an altimeter function (height above sea level) if it is calibrated against a known height before a climb. The project holds a meeting to triage the proposals, and decides not to reject the barometer goal.

The next task is to decide on an acceptance criterion to measure the barometer function; perhaps we will accept a reading accurate to within one millibar. A ‘priority to stakeholders’ of ‘mandatory’ completes the process of Figure 10.3 on the input side. The project may also develop scenarios, rationale and definitions (Chapters 5, 7 and 8) to clarify the barometer function.

On the output side, the task is to consider what design elements to include. Two design elements that are relevant include a pressure sensor, with software



Figure 10.4: Storyboard scenario for climber’s barometer function.

to calibrate pressure to altitude; and a GPS receiver, which can indicate altitude but not pressure.

- If GPS is chosen but not the pressure sensor, climbers will get a GPS altitude estimate (along with GPS positioning) but no barometer function.
- If the pressure sensor is chosen, users will get a barometer to help predict when the weather is changing, and climbers will get a barometric altimeter.

Perhaps these advantages justify the few grams in weight of the pressure sensor. Our team is confident that a handheld device can measure and display pressure (and temperature) accurately, as rival firm Bar-o-Graphics Inc sell a weather station for £99, though without the other features that we plan to include in our multifunction device. Table 10.1 shows how the requirement grew from a proposal to an agreed element of phase 1 of the development.

Dependencies

The prioritisation of individual requirements makes the assumption that they are independent of each other, i.e. that you can pick and choose any mix you like. That assumption is false when requirements depend on each other.

There are two main kinds of dependency:

1. **Scenario dependency:** two (or more) requirements occur only in the same scenario.

For example, to send colour facsimiles, you need colour imaging (e.g. a scanner) at the source, colour data transmission, and colour display/print

Table 10.1: Life history of a requirement.

Stage in life history (see Figure 10.3)		Requirement Attributes and Values	
Pre-optioneering 'Goal'	Proposed	<i>Parent goal</i> <i>Subgoal</i>	Predict changes in weather Barometer
	Draft	<i>Text</i>	Display barometric pressure
	Measured	<i>Text</i> <i>Acceptance Criteria</i>	Display barometric pressure Accurate to ± 1 millibar
	Input-prioritised	<i>Text</i> <i>Acceptance Criteria</i> <i>Priority to Stakeholders</i>	Display barometric pressure Accurate to ± 1 millibar Mandatory
Post-optioneering 'Requirement'	Output-prioritised	<i>Text</i> <i>Acceptance Criteria</i> <i>Priority to Stakeholders</i> <i>When to be Implemented</i>	Display barometric pressure Accurate to ± 1 millibar Mandatory Phase 1

at the recipient's end. There is no point providing one feature but not the others, as the scenario cannot then be completed successfully.

When a function is used in two or more scenarios, it should have the same priority as the highest priority scenario that uses it.

2. **Infrastructural dependency:** a requirement can only be implemented if another requirement for a piece of essential infrastructure is also implemented. (Infrastructure is often only hinted at in scenarios.)

For example, you can't provide an Internet-based traffic information service (say, goal C in Figure 10.5) in the luxury car telematics system you are designing, unless the mobile Internet access (infrastructure I) that the service depends on is also implemented. (If I has to be cancelled, A, B, and C must all be abandoned.)

In practice, infrastructure requirements are usually placed in the first possible release, whose design must form a stable platform for later additions. The Unix operating system, with a highly stable 'kernel' surrounded by successive 'shells' of software, is a good example of a well-planned infrastructure.

When an infrastructure requirement (like I in Figure 10.5) is used by two or more requirements (often these are functions), it should have the same priority as the highest-priority requirement that uses it. For example, in Figure 10.5, if A is a key goal and both B and C are needed to achieve it, then B and C must also have high priority. Even if C is only optional (i.e. A could be achieved by B and human effort alone) then I remains at high priority because of its relationship to B.

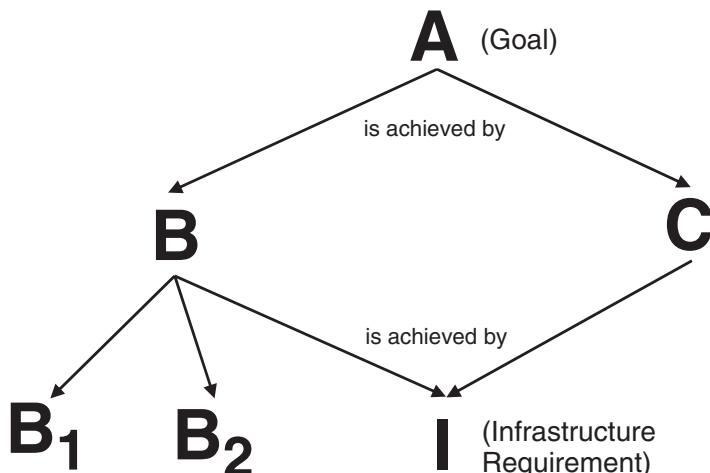


Figure 10.5: Behaviour of requirements with dependencies.

Requirements thus form dependency groups. Requirement prioritisation should be applied to dependency groups, not to individual requirements, to avoid creating inconsistencies during triage.

Needs or Features?

Churchill said 'Beware of needless innovation, especially when guided by logic'.
Richard L Wexelblat

When projects try to prioritise, they often consider features that could be added to a known design, rather than actual results asked for by stakeholders.

For example, the product manager for a new low cost, fuel efficient, lightweight car might be faced with a choice between incorporating power steering, electric windows and antilock braking. These are candidate functional features, found in many cars, that might or might not be worth incorporating in the new model.

Requirements theory would suggest that starting from features in this way is wrong, as you are supposed to start from stakeholders' needs. However, people's basic needs from a car – to get from A to B, safe and dry – do not help the product manager much. If features work for you, use them, but be aware that you may be trying 'push' features on to customers, rather than allowing them to 'pull' by stating their needs.

The risk in developing features because they are possible, or because someone in your own organisation thinks they might be interesting, is that customers may not want them. If you can validate your choice of features with stakeholders, do so.

Of course, pressures may come from other sources (e.g. all your competitors now have power steering in their rival products, so you have to include it too).

For example, using Figure 10.5 again, suppose that A is a key goal, achieved by subgoals B and C. B in turn is achieved by B₁, B₂ and I. I is also needed to enable C. Now, if B is deleted or postponed for budgetary or planning reasons, I must not be deleted as it is still needed by C, so the potential savings are limited to B₁ and B₂.

Dependencies mean that stakeholders need only state their (input) priorities for their goals; dependent subgoals and system requirements can be prioritised via their dependency traces to stakeholder goals. A goals workshop is therefore a good time to explore input priorities.

Notice that in these examples we had to look both ‘up’ towards goals, and ‘down’ towards infrastructure to avoid creating inconsistencies. Dependency networks can quickly become too complex to manage ‘by hand’, and are best handled with a requirements management (RM) tool. Such tools allow you to construct traces as database links between requirements, to indicate relationships of different kinds (see Appendix C).

Competing for Resources: Cost-Benefit Analysis

In some situations, requirements essentially compete for scarce resources. In the case of vehicles and mobile devices – with a spacecraft being an extreme example (see Chapter 3) – such resources include mass and electrical power.

Perhaps more often, the key resources are simply development time, skill and money. Risk is also important, because a more risky requirement may consume more resources than expected.

Competing requirements should be compared on both their expected benefits to stakeholders (i.e. their input priorities) and their ‘costs’, however these are measured (mass, power, risk, etc). The requirements can then be drawn on a Boston Matrix ([3] Figure 10.6) to help your team agree which have the most favourable benefits compared to their costs.

Where benefits are directly quantifiable, you can have actual values on the ‘benefit’ axis of your Boston Matrix, and you can compare requirements by their calculated benefit/cost ratios.

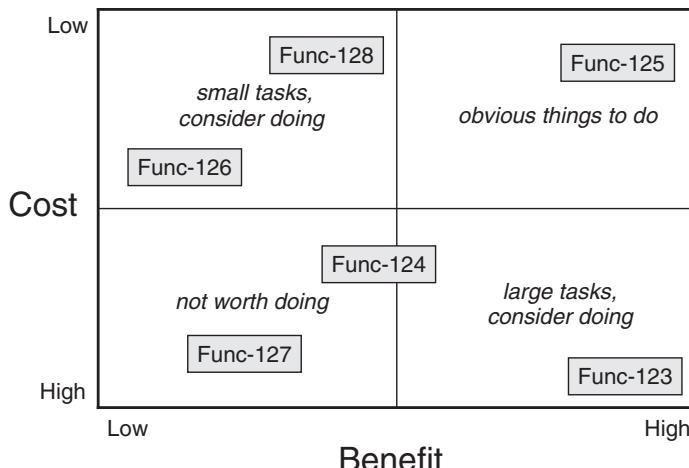
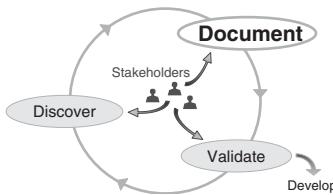


Figure 10.6: Boston Matrix of benefits and costs.

10.4.2 Documenting Output Priority



If given a list of requirements with traditional priorities (e.g. ‘mandatory’, ‘desirable’, ‘possibly useful’), many projects only develop the requirements with the highest output priority (e.g. ‘mandatory’), at least in their first cycle of development. That results in ignoring many good and worthwhile requirements, forcing stakeholders who have experienced such things to demand ‘mandatory’ status for all their requirements. That makes prioritisation difficult.

Therefore, you may find it helpful to think of output priority as stating the time (perhaps your programme phase) when each requirement should be implemented.

The ‘Cherry Stones’ Approach

There is an old lover’s game played with cherry stones.³ The would-be lover eats the cherries and lines up the stones on the side of the plate. The stones are then counted off in fours with the rhyme:

‘[He/she will marry me] this year, next year, sometime, never.’

Applying this to prioritisation, we get a triage scheme as follows:

- Implement it in the current phase (i.e., ‘triage it in’).
- Implement it in the next phase (or in the current one, if we have time).
- Possibly implement it in some future phase.
- Discard it as impracticable (i.e. never: ‘triage it out’).

The results of such triage (in a process much like Figure 10.3) can be documented straightforwardly as a requirement attribute, e.g. {Phase 1, Phase 2, Future, Never}, as shown in Figure 10.7.

For a truly minimalist ‘agile’ approach, on a project where you don’t need to plan much of a detailed roadmap, you could perhaps use just {Now, Later}.

³Also known as cherry pits or seeds, though the seed is actually the edible part inside the stone’s shell.

ID	Function	When to be Implemented
Func-321	Print an address label.	Choose a value: ▼
		Phase 1
		Phase 2
		Future
		Never

Figure 10.7: Documenting output priority with a single attribute.

The 'Product Roadmap' Approach

However, if you expect to have:

- several phases of development (e.g. you will release updated versions of your car in March and September this year, and in March next year);
- product variants (e.g. your car has versions for the UK, USA and Japan markets); or
- a product line (e.g. you are making a basic L, a medium XL, and a luxury XLS version of your car);

(where several different product releases share some of the requirements), then it is probably simplest to have one output priority attribute per phase. You can then construct a convenient table of applicability of requirements to phases (Table 10.2). With a requirements database tool or spreadsheet, this allows you to filter only the requirements that are essential in Phase 2, etc

Product variant priorities can be recorded in similar style (Table 10.3). Again, a given requirement may have several output priorities, in this example according to the geography of the product's market.

When you have both multiple product releases and product variants, as is likely if you are working on a product line, you will need to combine the effects of Table 10.2 and Table 10.3, for example by linking two or more tables in a requirements database tool.

Table 10.2: Documenting output priority for multiple product releases.

ID	Function	Output priorities		
		Phase 1	Phase 2	Phase 3
Func-125	Play MP3 music files.	If possible	Essential	(Already done in Phase 2)
Func-126	Play iTunes.	-	If possible	Essential

Table 10.3: Documenting output priority for product variants.

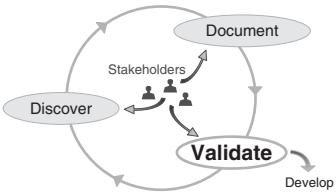
ID	Function	UK	USA	Japan
Func-127	Show roadworks locations.	Desirable	Optional	Essential
Func-128	Show fuel locations.	Desirable	Essential	Optional
Func-129	Show traffic camera locations.	Essential	Desirable	Optional

A Requirement Manager's Lists

Requirement managers can handle triage with a combined specification and 'waiting list'. Requirements tools and spreadsheets allow tables to be filtered on list values such as those in the 'When to be implemented' attribute in Figure 10.7:

- Filtering on 'Future', for instance, displays the waiting list.
- Filtering on 'Phase 1' yields the list of requirements for the current cycle of development.
- The implication is that you have to filter before printing.

10.4.3 Validating Output Priority



Setting output priorities means choosing which way to go on your project. This is inherently risky as you are predicting what will work best, given the realities of your world (the skill, time, money and facilities at your disposal; the state of the market; the way the design you have chosen will actually work).

You can take action to:

- understand the effects of your requirements on design, as discussed in Chapter 14;
- check that your choices are likely to work, by:
 - prototyping
 - market analysis and survey

- competition analysis
 - risk analysis
 - review;
- **check that the set of requirements or features chosen for a release makes sense**, in terms of:
 - scenario and infrastructure dependencies
 - benefit to product users
 - compatibility of timescale.

A Product Manager's Roadmap

A product manager can see at a glance from Table 10.1 that Func-125 is planned for Phase 2 but might, with luck, be implemented earlier, in Phase 1. If all goes according to plan, it should already be working in the product when Phase 3 starts.

The table of requirements with their output priorities by phase can in this way act as a programme roadmap.

The phases differ from stand-alone projects in:

- sharing many of their requirements;
- being for successive versions, releases or models of a product;
- often running partly in parallel, i.e. Phase 2 may start before Phase 1 ends.

If these show that risk remains high through uncertainty about the design, you should change your development life cycle to one explicitly constructed to control risk [5]. For example, you could build a demonstrator for each of two competing design approaches, evaluate them in trials, and only then make a final choice as to which one to take into service.

10.5 Things To Check Priority Against

- Goals (Chapter 3)
- Dependencies via traceability
- Priorities mentioned in interviews (Chapter 11), workshops (Chapter 12), business plans, vision statements, etc
- Trade-off decisions (Chapter 14)

10.6 The Bare Minimum of Priorities

Agree which requirements will realistically be done in the first phase of the project.

10.7 Next Steps

- Review your requirements. A suitable review process is described in [4].
- If you haven't already done so, plan your test campaign in relation to the requirements. Your scenarios and acceptance criteria (see Chapters 5 and 9) should be your starting points.
- Put all the requirements together, make a read-only 'baseline' copy of them with a suitable name (e.g. 'ABC Requirements v1.0') and publish them to the project (Chapter 15).

10.8 Exercise

List the input priorities for the multifunction device for outdoor leisure use (for which you developed scenarios in the exercises in Chapter 5). Would all the stakeholders agree on these?

10.9 Further Reading

Prioritisation is not well covered in the requirements literature, and research is only just starting to identify techniques that might be practical in industry. Two books that do address it are Davis [1] and Goldsmith [2].

10.9.1 Triage

1. Davis, A.M. (2005) *Just Enough Requirements Management*, New York: Dorset House.

Al Davis has written an insightful, funny book that anyone who runs requirements workshops will find stimulating. The book suggests many practical techniques for triage and the balancing of multiple factors including cost and risk.

10.9.2 Input Priority

2. Goldsmith, R.F. (2004) *Discovering REAL Business Requirements for Software Project Success*, Boston: Artech House.

Robin Goldsmith's book provides a wonderfully practical description of things to do to determine whether proposed requirements are really necessary.

10.9.3 Boston Matrix

3. http://www.marketingteacher.com/Lessons/lesson_boston_matrix.htm
The Boston Matrix is widely used in business. See this website for an example.

10.9.4 Review Process

4. Alexander, I.F. and Stevens, R. (2002) *Writing Better Requirements*, London: Addison-Wesley.

A standard review process is described in Chapter 8.

10.9.5 Life Cycles

5. Farncombe, A. (2004) Project Stories: Combining Life-Cycle Process Models. In Alexander, I. and Maiden, N. *Scenarios, Stories, Use Cases*, Chichester: John Wiley & Sons, Ltd.

In Chapter 15, Andrew Farncombe describes how to select development life cycle approaches appropriate to different situations.

PART II

Discovery Contexts

<i>Requirement Elements</i>	Priorities	Measurements	Definitions	Rationale	Qualities and Constraints	Scenarios	Context, Interfaces, Scope	Goals	Stakeholders
<i>Discovery Contexts</i>									
Introduction									
From Individuals									
From Groups									
From Things									
Trade-Offs									
Putting it all Together									

This part of the book describes the contexts within which you can effectively discover requirements. These include the traditional environments in which consultants meet stakeholders – interviews and workshops – as well as consulting the public, observation, ‘reverse engineering’ and reuse. It also describes how to make the transition from discovery to using the requirements to drive your project. This includes trading-off what is wanted against what can be achieved.

CHAPTER ELEVEN

Requirements from Individuals

Paying serious attention to the application domain means writing serious, explicit and precise descriptions. Don't think it's good enough to interview a few people and make a few rough notes. Don't say: 'Everyone knows what you do when you book a theatre seat', or 'Everyone knows that the seats become available when the show is scheduled'. What everyone knows is often wrong, and always very far from complete.

Michael Jackson

Requirement Elements	Priorities	Measurements	Definitions	Rationale and Assumptions	Qualities and Constraints	Scenarios	Context, Interfaces, Scope	Goals	Stakeholders
Discovery Contexts									
Introduction									
From Individuals									
From Groups									
From Things									
Trade-Offs									
Putting it all Together									

Answering the questions:

- What do stakeholders individually do in their work?
- What do they individually want?
- What do they think of a proposed development?
 - ... so you find out people's points of view
 - ... so you learn what really happens.

11.1 Summary

There are two major ways of obtaining requirements from individual stakeholders:

- interviewing – planned meetings in which you ask questions;
- observation – planned sessions in which you watch, listen and learn.

Both of these have many useful variations. Interviews range from formal and heavily structured, to informal and open ended. In observation, you may be a passive, silent observer or an active participant. This chapter discusses how to apply the techniques to discover requirements from individuals effectively.

11.2 Introduction

There are as many ways of discovering requirements from individual people as there are styles of conversation and ways of living. Informal conversations with a stakeholder by the coffee machine, or with the team over a nice meal in a restaurant, can be extremely effective in finding out things left unsaid everywhere else.

There are traditional methods, too. A generation ago, analysts used to write questionnaires and send these out to anyone they thought might be affected. People often ignored them, or did their best with the slightly off-the-mark questions and columns of little boxes. It was never very clear how new requirements could be discovered in this way. Questionnaires are difficult to construct, arduous to process and leave little room for individuality, though they do have a role in the mass market and for public consultations (e.g. for a tram route).

Leaving these aside, the two major ways to discover requirements from individual stakeholders are:

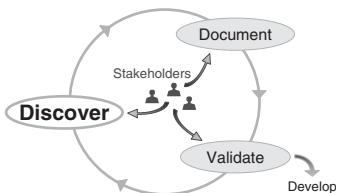
- interviewing (section 11.3);
- observation and apprenticeship (section 11.4).

Both involve spending time with a stakeholder. The main difference is that in an interview, you lead by asking questions, with the goal of encouraging the stakeholder to describe his or her requirements; whereas in the various forms of observation, the stakeholder leads by working as usual, or by carrying out agreed tasks.

Interviewing may appear the more direct route to requirements, but it depends on people's ability to describe their needs explicitly and accurately. There is plenty of evidence that many people find this difficult.

Observation is potentially a far richer source of information, but it does not necessarily reveal requirements directly. Some forms of observation provide you with a spoken interpretation as well as the chance to observe: the stakeholder (or one of their colleagues) may provide a running commentary, or you may be able to ask questions during or after a period of observation. In 'apprenticeship', you act as a beginning apprentice learning a task from the stakeholder. Such variations form a spectrum of useful approaches between pure interview and pure observation. Let us look at these two groups of techniques in detail.

11.3 Interviews



Interviewing is probably the most commonly used context for discovering requirements. It is a powerful tool but it also has disadvantages. In particular, it can be a poor way to understand processes that involve many stakeholders.

Neither unprepared nor fixed framework interviews are likely to work well.

Interviewing requires skill and planning. Requirements interviews are best conducted using two interviewers. A follow-up interview or other means of checking interview findings, such as a workshop, is advisable.

11.3.1 Planning an Interview Campaign

Interviewing stakeholders is the most direct way to find out what they want. However, interviewing brings both advantages and disadvantages. When planning, do not assume that interviewing is the right or only possible approach. You may need to combine interviewing with other techniques.

Advantages of Interviewing

Engaging with Stakeholders

The first advantage of an interview is that you can give immediate personal attention to the interviewee.

Stakeholders are nearly always pleased to be heard. Your first duty is to listen carefully and to engage in a friendly way. You will almost certainly hear things you didn't expect; your interviewees are the experts in their field, not you.

It is vital for stakeholders to see that what they say translates into action. You need to show them that:

- they are being heard;
- what they said is incorporated in requirements;
- their requirements turn into products and services.

Then, they will be eager to talk to you.

Dialogue and Feedback

Crucially, an interview allows you to ask as well as to listen. You should use the opportunity to feed back your understanding, giving the interviewee a chance to correct what you said, or to add to it. This should enable you to discover a stakeholder's point of view, and to check it with him or her straight away.

Follow-up

A second or follow-up interview is always wise, as it lets you:

- check your understanding once you have written up your findings;
- fill in any gaps;
- ask new questions suggested by findings from other interviews.

If you can't hold a follow-up interview, at least provide a draft of the interview for the interviewee to check, correct and sign. That takes time, so include enough time for checking back with stakeholders in your plan.

Disadvantages of Interviewing

*If I'd asked people what they wanted,
they'd have said faster horses.*

Henry Ford

Only What They Know

People are all different. Many interviewees are not at all accustomed to thinking outside the fixed framework of their day-to-day work. They can see,

maybe, that their work would be more comfortable with a bigger screen, or that some tasks could be improved with a slicker user interface. They may be unable to imagine a wholly new way of doing things, as Henry Ford remarks. It is hard to get people to devise really innovative products. Prototyping (see Chapter 13) and workshops (see Chapter 12) are part of the answer here.

Knowledge we can Show but not Tell

One of the obstacles to discovering requirements is that stakeholders may know something that ought to be documented, but they don't tell you because they can't put it into words: the knowledge is tacit.

Michael Polanyi (1966), in his book *The Tacit Dimension* [6] put the problem like this:

'We can know more than we can tell.'

There are several reasons for this.

One is that we know some things just by doing them, like riding a bicycle or using a tool. How hard should you pull on the handlebar to keep the cycle upright? Just enough to make it turn gently the other way. Your hands and body have the knowledge, not your 'head'. Unless you are a physicist you probably can't explain the skill, but you can show or teach someone to ride a bicycle. Observation and 'apprenticeship' – getting people to show and to teach – are described in this chapter.

Another example is facial recognition. How do you recognise your mother? You don't know, you just do. The question is directly relevant to software; for instance, how the police get witnesses to describe criminals. Direct questions (like: 'What was the robber like?') get vague answers ('Tall, stocky, he had short hair'). The witnesses know but can't tell.

A better way is for the police to show pieces of photographs – an ear, a nose, a chin – which a technician can fit together to make a likeness of the robber's face. 'Yes, something like that,' says the witness, 'or maybe a bit bigger'. The police technician patiently finds a larger chin. 'Yes, that's him,' says the witness.

Notice that the mystery does not go away: the witness still does not know how she recognised the robber's chin, and nor does the technician.

But something useful has happened: through an appropriate knowledge discovery technique, a recognisable likeness of the wanted suspect has been created by the witness/technician team. A description has not been spoken, but one has been discovered.

Prototyping (Chapter 13) and scenarios (Chapter 5) also exemplify discovery through showing rather than asking.

'One Hand Clapping'

A major disadvantage of interviewing is that you are only seeing one point of view at a time. This does not matter when that person carries out a whole process. But when each stakeholder only carries out some isolated steps in several processes, interviewing one person at a time is rather like hearing one hand clapping. In that case, a scenario workshop (see Chapters 5 and 12) is more efficient and less confusing.

Unresolved Conflicts

Another disadvantage of interviewing is that conflicts are unresolved. Stakeholders do not get to hear each others' points of view. Your analysis and understanding may enable you to detect conflicts. You may even be able to see possible ways of resolving those conflicts. But the stakeholders themselves will not. Again, a workshop may be the answer. Certainly, a dialogue will be needed. (See Chapter 2, and Section 3.2.6, The Negative Side in Chapter 3.)

How many Interviewers, How many Interviewees?

The one-to-one nature of an interview is both a strength and a weakness.

Two Interviewers

Requirements interviews can definitely benefit from two interviewers:

- one to ask most of the questions;
- the other to do most of the recording, and to watch for possible missed requirements.

The lead interviewer can also write notes; the other interviewer can also ask questions. This sounds expensive, but the effect is to have two pairs of eyes and two pairs of ears on the job. One missed requirement costs as much as ten requirements discovered early. Working in a pair is more effective. Over the life of a project, it is also cheaper.

Interviewing Several Stakeholders at Once

It is possible to interview two or three stakeholders at once. But you don't want an interview to involve so many people that you lose control, and the dialogue breaks down into a general discussion.

Sometimes, people in one department suggest that they should attend together when you invite one of them to an interview. In this case, it is best to accept. They probably share the same point of view, and can contribute details to give you a fuller picture of their work.

Never Interview People with their Boss

Interviewing people together with their boss or department head is less satisfactory, because either:

- people are shy in front of their boss; or
- the boss insists on speaking for the department.

Either way, you won't find out what is really needed. Bosses often spend more time on administration than on technical work – that's their job. Even if they used to do a technical job, they may have a rusty idea of the work.

Following the Rule Book?

We once had the belief that in a job that was all about safety, the rule book would reign supreme. If you asked some air traffic controllers to explain what a colleague was doing in any particular operation, we supposed you would get an answer like 'Runway Operations, section 7.4 paragraph 1'. They would be so well trained in safety procedures that all they ever did was to apply the right procedure to the situation at hand.

On meeting some engineers who worked in air traffic management, we shared our thoughts and asked if that was how it was. They laughed merrily.

'It's nothing like that at all. At any moment, there are probably five rules that apply. We have to decide which rule to break, or we'd never get the planes down at all.'

Moral of the story: what happens in the real world is *never* exactly what is written down. A manager in an aerospace company had a phrase for trying to describe every detail of product development in written procedures. It was 'boiling the ocean' – a task unlikely to be completed any time soon.

Another hazard is the boss who knows the rule book:

'The usual procedure, as defined in section 3.2.1, §4, is to . . . '

Unfortunately, that is almost certainly not what anyone actually does. Finding out what the person who wrote the rule book once thought was what ought to happen is not the same as finding out:

- what does actually happen; and
- what actually goes wrong.

So, if the boss wants to be interviewed, do that one separately. But also insist on seeing the people who do the technical work. If the boss can't see the point of that, say that you'd like to capture all points of view, including details of day-to-day operations.

Time Needed

An interview campaign takes time, for several reasons:

- As a rough guide, allow one hour per interview. Be ready for interviews to take longer – do not think of this as overrunning, but as a chance to discover more from key stakeholders. Conversely, some interviews are all over in a few minutes.
- Allow at least two to three hours to document, compare notes, analyse and trace your findings.
- You may have to travel to and from interviews; many organisations are split over several sites.
- Allow time to prepare, by finding out what you can about the interviewees and selecting suitable materials to stimulate them to talk.
- Interviewees are invariably busy on other work. Your project and interview campaign may be minor concerns to them. It may take you time just to get through to arrange an appointment. The interview slot may be weeks ahead.
- Managers, especially, may cancel interviews at short notice. Be patient and reschedule. Allow time simply to manage your interview schedule.

These factors limit you to about two interviews per day, on average.

Can it be Done More Quickly?

Interviewing as many as four people in a day and analysing the findings effectively is possible in a crisis, but it is very hard work.

Tips for Planning Interviews

- Get management support.
- Plan your interviews, but be willing to change your plans.
- Find out about your interviewees in advance.
- Get some materials, e.g. existing models, scenarios, prototypes or statements made by other interviewees, to stimulate the interviewee to respond.
- Look at existing documentation and systems.
- Talk through your interview plan with a colleague.
- Allow time for follow-up interviews, writing up, and analysis.
- Allow time for interviewees to correct and sign the notes of their interview.

If you have to get results quickly, you require clear support from senior management. Once your stakeholders know that you have high priority, appointments can be made and held quickly.

Sometimes, a department's administrator has authority to put appointments into people's calendars. If you can get the help of an efficient administrator, your interview campaign will go far more smoothly.

You may possibly be able to save time by interviewing two or more people at once (see the earlier section headed 'How Many Interviewers, How Many Interviewees?'). However, that changes the nature of the interview.

11.3.2 Planning Each Interview

The plan is nothing; the planning is everything.

Dwight Eisenhower

All interviews are different. They are human interactions, and need both planning and flexibility:

- A rigidly planned formal interview, in which you follow a script, is not likely to help you create good requirements.
- Conversely, an unplanned informal interview, in which you turn up with no idea what to ask, is not likely to work well either.

Eisenhower's is the right approach: think hard when planning, then be quite relaxed about departing from your plan. Before the interview, have a sharp idea what you want, who you are interviewing and what you might ask. Then in the interview itself, you are ready to use your plan or let the interviewee steer the dialogue. When an interview flows naturally, the requirements usually surface without effort.

What to Ask: The Six Journalist's Questions

If you are not sure what to ask, think of yourself as a journalist trying to put together the facts for a story. The traditional advice to new journalists is to use Rudyard Kipling's six 'honest serving men':

'What and Why and When and How and Where and Who'.

For example, the business is introducing a new process. You ask (preferably not in exactly these words – you should adapt them to each situation):

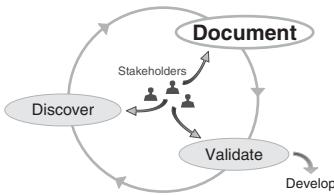
- What is the new process?
- Why is it being introduced?
- When will it be rolled out?

- How will it work?
- Where will people find it?
- Who is to use it?

Do not try to use such questions mechanically – it won't work. Instead, use your preparation and knowledge from earlier interviews to guide your questioning intelligently. Interviewees' answers will lead you into more detail, which will suggest more questions to you as you go along.

As a journalist, you will also want to check your facts, to follow up leads that may reveal important new stories and, if possible, to obtain a 'scoop', a genuine new discovery.

11.3.3 Documenting Interviews



Note-taking

The traditional way to document an interview is to make notes. With practice, this can be an effective way to create requirements. Handwriting is slow, and talk is fast. So, your only chance of catching the essence of what is said is to think – clearly and continuously. You then write down a short summary. Some colleagues compare this to panning for gold: you let the water wash away the sand, and just pick out the nuggets of pure gold when you see them.

Note-taking therefore requires you to do four things all at once:

- listen carefully;
- think clearly;
- write crisply;
- ask for clarification when necessary.

The difficulty of doing this is a major reason for sharing the work with a colleague: two heads are better than one.

Drawing Diagrams 'In Real Time'

Sometimes, making a simple model is more effective than making traditional notes. A scenario is an obvious choice, but mind maps [9] (see Appendix C), flowcharts and goal models can also be helpful.

Tips for Interviewing

- Start by introducing yourself and saying what the interview is for.
- Be friendly and professional.
- Don't assume stakeholders know anything about your project.
- If possible, work in a team of two:
 - one person does the interviewing;
 - the other (the 'scribe') records what is said, and observes the interviewee's reactions.
- Do treat interviewees as expert in their own field.
- You have two ears and one mouth; listen and speak in that ratio.
- With a shy interviewee, be encouraging, and give them something to comment on to 'break the ice'.
- If a stakeholder says something three times, it's important!
- Do not force your own opinions, feelings and ideas on the interviewee.
- Use your interview plan as a guide; depart from it when necessary.
- Don't lead the witness; use neutral questions like:
 - 'Can you tell me more about ... ?'
 - 'Could you talk me through how you do that?'
- Explicitly check your understanding, e.g.:
 - 'If I've got this right, you first do X and then ... '

The goal of 'modelling in real time' is not to make elegant diagrams. It is to understand the structure of what your interviewee is saying. Above all, never try to show how clever you are, or how much you know about modelling languages.

For example, when you understand that your interviewee is describing a process, you can decide to write a scenario or draw a flowchart. You could introduce that by saying: 'It sounds as if there's a process here. Could you describe it step by step?'

With some interviewees, especially if they have an operational role with a system, the result is that they quickly start listing the steps they take to achieve a task. You will then find it easy to document their process.

With other interviewees, especially if they are only involved in one step of a process, or if they do not work in operations themselves, you will not get such a clear result. They may describe several different cases or situations. They

may wander off into an account of some issue or other. When this happens, you should quietly drop your scenario or flowchart approach.

Types of model most often useful in requirements interviews include:

- stakeholder maps or onion models (see Chapter 2);
- goal models (see Chapter 3);
- rich pictures (see Chapter 4);
- flowcharts or other scenario pictures (see Chapter 5).

Other types of diagram should be chosen with care, depending on the sort of person you are talking to, and any special purposes your project may have.

Types of model useful in special circumstances in interviews include:

- rationale models, showing assumptions to justify project decisions (see Chapter 7 and Figure 11.1);
- organisation charts (organograms);
- Venn diagrams;
- decision tables;
- models of the development life cycle;
- mind maps [9];
- goal models purely of nonfunctional requirements (NFRs);
- sketch maps (e.g. of transport infrastructure);
- sketched building layouts;
- information models (for databases).

Presumably, almost any kind of model or diagram can be useful at some time or other.

Some analysts habitually draw dataflow diagrams; others always draw data structures using traditional entity-relationship diagrams or modern class diagrams. Personal experience and preferences seem to be important here.

The key thing is to use techniques that both you and the interviewee are comfortable with. Be relaxed with any model you use. Present it as a simple and natural aid to communication. Your interviewee will then talk to you about the things the diagram is describing, rather than the diagram's notation.

Michael Jackson (1995) [2], famous for his 'Jackson Structured Design' method, says of analysts who only use one method or type of model:

‘To the man who only has a hammer, every problem looks rather like a nail.’

In ‘real time’ contexts like interviews, you must often improvise. In an interview, you don’t have time to start looking things up. This means that to create

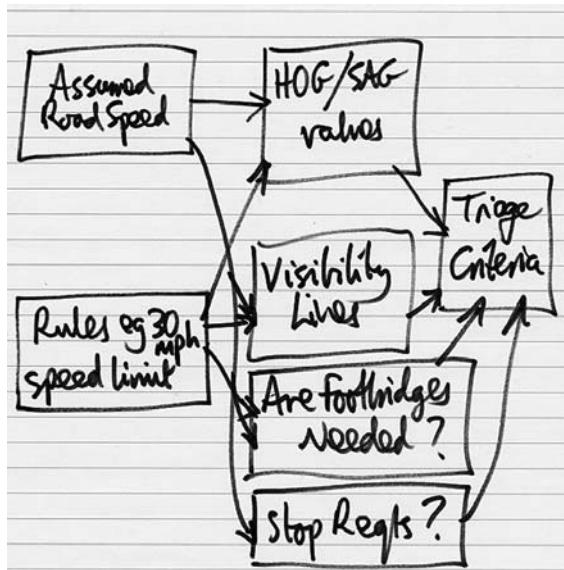


Figure 11.1: Rationale sketched in notebook during an interview.

good requirements, you need to be familiar with a range of techniques. These should include both the requirements discovery techniques described in this book, and the standard analysis techniques (statecharts, class diagrams, etc) that you'll find in Martin Fowler's (2004) *UML Distilled* [7] or any good introduction to software engineering, such as Ian Sommerville's (2006) *Software Engineering* [8].

Using a Computer in an Interview

Both text and diagrams can be created on a computer. Laptop computers are easy to carry. Are they therefore ideal for interviews?

In general, the answer is no, they're not. If you are looking at your computer and thinking about icons, pop-up windows and so on, you are not paying full attention to your interviewee. It's even worse if you're fiddling about with some complicated drawing tool, trying to open the right windows and arrange boxes neatly in a row. You are spending all your interview time 'in the computer'.

Some people, maybe you, are incredibly quick at drawing models with a computer. In this case, the danger is not delay, but leaving the interviewee behind. You are so dazzling with your SysML models that the interviewee falls silent. Your world is too far from theirs, and you miss their requirements.

So, the rule is to be 100% present in an interview (no PDA, no mobile phone that goes off in your pocket, no computers).

But rules are made to be broken:

- You may want to show your interviewee a diagram or photograph and ask for comments. A computer is ideal for this. You may then want to respond to a comment by immediately editing the diagram slightly.
- Or, you may be wonderfully quick at touch-typing on a nice quiet laptop.

If you can record quickly and without fuss – that is, without interrupting the flow of the interview – then that is fine, though you should still weigh up the risk of appearing as a ‘techie’. It is, by the way, far easier to record unobtrusively if you are interviewing as a team (normally two of you) rather than on your own.

Recording Interviews

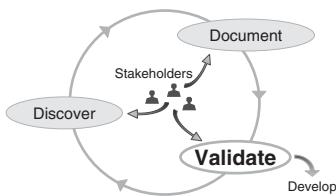
An audio (or even a video) recorder may be useful, but remember that:

- many interviewees do not like to be recorded;
- transcribing audio recordings is time-consuming: plan for it;
- you need written permission to record in some organisations;
- relying on playing back a recording after the interview is far less likely to clarify a garbled requirement, or to reveal a missed one, than simply asking the interviewee for clarification after the interview, whether in person, by phone, or by email.

Tips for Documenting Interviews

- Record who is present, their roles and organisations, and the date.
- Listen.
- Write down questions in the interview, and ask them later. Putting an '*' or '?' in the margin is helpful, as it makes it easy to locate things you need to clarify.
- Number the statements you record. This helps you to listen attentively at the time. It also helps you to identify and trace requirements later.
- Make sure you understand what you are writing.
- If anything is unclear, ask.
- Don't hurry. If an interviewee is going too quickly, ask him or her to give you a little time, and read out what you are writing down. If need be, ask them to confirm that you have written it down correctly.

11.3.4 Validating Interview Findings



There is not a great deal to say about validation that is specific to interviews, as you can use the validation techniques from every chapter in Part I to check your findings.

Beginner's Mind

*In the beginner's mind there are many possibilities,
but in the expert's there are few.*

Shunryu Suzuki

Your own understanding is incomplete, and wrong in many ways. This will probably be obvious to you at the start of an interview campaign.

More effort is needed to go on with what Zen masters like Shunryu Suzuki [10] call 'beginner's mind', that desirable state of openness and willingness to learn that comes naturally when you know that you don't know much.

Checking your Understanding

Checking your understanding may involve revisiting anything you have done or documented.

In the interview itself, you can conversationally check from time to time that you have understood things. This can be somewhat random; for example, you can check you have a right understanding of the trickier points – you certainly don't want to seem to be challenging every single statement. Or, you can wait until a section of the interview is complete, and more deliberately sum up what you think you have heard, inviting the interviewee to correct you.

Getting a Second Opinion

You can also use interview outputs – notes and diagrams – to stimulate other interviewees. This automatically gives you an independent opinion on interviews you have documented, and helps to fit the interview findings together.

Validation by the Interviewee

After the interview, it is sensible to allow the interviewee to validate your findings directly by signing – and if necessary, ‘red-lining’¹ – a copy of the interview notes.

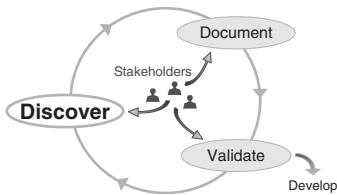
Tips for Validating Interview Findings

- Start in the interview itself, by checking back with the interviewee.
- Use the findings to stimulate other interviewees.
- Check the findings against the other things you have discovered (dictionary, goal model, scenarios, etc).
- As soon as possible after the interview, give your interviewees the opportunity, to check what you think you have discovered from them, firstly by reading your notes of their interview, and later by reviewing the requirements.

11.4 Observation and ‘Apprenticeship’

Observing work, by any of several different techniques, can help you understand a problem better than any amount of interviewing. Observation is relatively little used for discovering requirements, but can be both quick and effective.

11.4.1 Making Observations



Introduction

What you Can See

Yogi Berra remarked that you can see a lot just by looking. Where you are trying to understand the requirements of a group of skilled professionals, a

¹Correcting by marking up with a red pen, literally or with a word processor’s ‘track changes’ function. Since a signature is desirable, it may be best to ask for corrections on a paper copy.

short period of observation is a good way to start to understand the working environment. Observation is also called ‘ethnography’ or ‘fieldwork’, as you leave your office to see what actually happens ‘out in the field’ [5].

For example, air traffic controllers and financial traders work rapidly in a busy environment full of data and other people. Both groups use specialised language, communicate in brief bursts, and handle many problems at once in ‘real time’. What you can observe will be very different from an abstract written description.

You may find it helpful to watch both an experienced user and a novice. That will reveal very different things about using a tool.

What you Can’t See

You should not expect to discover many new requirements simply by observation.

Firstly, important kinds of event may be rare. You might have to watch for many years for a chance to observe them. Many exceptions, especially, would be serious if they occurred, but occur very infrequently.

Secondly, what you can see is a situation, to which a requirement might perhaps be relevant – if the team can identify it. Identifying requirements takes creative effort, and as Torvinen suggests,² that effort needs to be collaborative.

Observation Benefits

Observation is good for:

- getting the feeling of the working environment;
- discovering practical issues;
- stimulating dialogue with your stakeholders;
- seeing that a given approach won’t work.

Observation Techniques

For requirements work, simple observation techniques useful for creating requirements include:

- silent observation;
- being ‘talked through’ operations;
- being coached or ‘apprenticed’.

Silent Observation

Spend short periods of observation (say, five to ten minutes) sitting silently ‘behind the shoulders’ of your stakeholders. This may be necessary in an

²‘Requirements cannot be observed or asked for from the users, but have to be created together with all the stakeholders.’ – Vesa Torvinen. We discussed this in Chapter 1.

operational environment like a control room or a financial dealing room, for example.

If you have any questions, note them down and ask them afterwards, outside the operational environment.

A short video recording could be valuable in this context, if you can get permission to make one. It may give the ‘flavour’ of the work, which may be difficult to describe any other way.

Silent Observation of Use of Prototypes

Silent observation can be eye-opening in combination with prototyping (see Chapter 13). You may observe that users run into trouble in places you did not expect when trying to use your prototype. You may see that they expected a different arrangement of the user interface, or were attempting a sequence of operations (a scenario) that you did not know about. Such observations quickly lead to new requirements.

11.4.2 Being ‘Talked Through’ Operations

Get a practitioner to talk you through some operations while he or she carries them out. It is often quite difficult to understand what someone is doing when they are hammering away rapidly on a computer keyboard or clicking on menus, for example.

Fortunately, practitioners can often give you a running commentary on what they are trying to achieve, and what they have succeeded in doing.

Again, an audio or video recording, or still photographs, may be useful to you later. Always ask in advance for permission to make any kind of recording.

Being Coached or ‘Apprenticed’

Arrange to try a task yourself, with a practitioner coaching you. You are immediately in the position of *a beginning apprentice*, starting to learn how to do the work. That puts the practitioner in the role of *master craftsman*. He or she will naturally teach you what to do, not by lecturing to you intellectually but by showing you the work, a step at a time. This means you see the work from the inside, and you get to know it as a practice, not as a theory. Apprenticing works especially well with skills that are hard to explain.

In the case of critical or safety-related work, you may be able to work on a training simulator, or perhaps use a live system on old or simulated data. Again, this is much more effective if a skilled practitioner can sit with you and ensure that you do things the right way.

Traditional apprenticeships lasted seven years. Fortunately, even very short periods of apprenticesing can be effective. You could plan a series of three one-hour sessions, for instance. In the first session, you will mainly be feeling

your way, and everything will be confusing and unfamiliar. By the third session, you will be starting to acquire some basic skills and understanding.

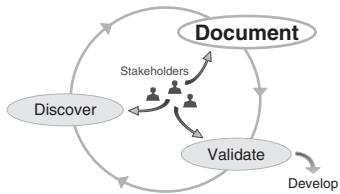
Apprenticing is a good way of getting to understand the 'look and feel' of a system, and to respect the skill of practitioners. It also gives you an opportunity to ask questions about the system and what people would like to see improved. It is another way to practise 'beginner's mind' [10].

Since you will not be able to write notes while doing this, you should have a colleague at hand to observe and record what is said. Do not assume that your feelings or experience are representative of those of skilled practitioners.

Tips for Making Observations

- Ask your stakeholders if it would be possible to observe their work for a short period.
- Discuss with them which techniques would be most appropriate.
- Obtain agreement (in writing, if necessary) if you want to use a camera or voice recorder.
- Consider having two people observe the same scene, with only one of you making notes.
- Debrief afterwards, with the stakeholders if possible or, if not, then amongst yourselves.
- Ask questions, as soon as possible after the observation period.

11.4.3 Documenting Observations



Treat observation time as special, like going to the theatre or a concert. Prepare for it by learning about what you are going to observe. Plan how you will record your observations; observing and recording simultaneously is not easy.

Recordings may well contain sensitive information. Take care to keep your recordings confidential. Label your original media (tapes, image files, etc) with the details of the observation session: date, place, people involved. Preserve

the original media, boldly mark originals as such, and always work on copies. You may need to protect the identities of people you observe.

Notebook

Handwritten notes are the traditional means of recording field observations. A notebook and pen permit you to:

- write in longhand, i.e. to make deliberate observations in carefully constructed sentences;
- make jotted notes using any kind of shorthand (official or personal);
- support your notes with sketched diagrams;
- keep notes together in a book, indexed by date.

Handwriting is quiet, unassertive (unlike opening a laptop) and reliable (it needs no batteries). It does not depend on bringing delicate and expensive equipment into the field. For all these reasons, it is likely to remain a useful medium.

Computer

We have already stated our reservations about using a computer in an interview. Some of these also apply to observation.

You may possibly be able to make observations and record them simultaneously on a laptop computer. Since you should be watching and listening, you need to be able to touch-type quickly and accurately so you will not be distracted by looking at the keyboard or screen.

The usefulness of a computer for note-taking depends on the length of the period of observation. For very short periods (say, five to ten minutes) there seems little reason to try to get notes into the computer 'in real time'. For long periods, you need to have a machine with enough battery life to allow you to work uninterrupted.

At all costs, do not let the task of editing a text interfere with the work of observing. You can always tidy a text afterwards, but the brief moment of observation may never be repeated.

Audio

There are two basic possibilities for audio:

1. Recording events as they are: you simply switch on the recorder and listen. A 30-minute session creates a 30-minute recording. This is useful only in domains where what people say tells most of the story of what is happening. You could supplement your recording with still photographs to show the scene at different stages.

2. Making 'audio notes' by speaking softly into the microphone yourself. A voice-activated dictation recorder, i.e. one that only records when you speak, could be convenient.

It may be possible to combine the two techniques, i.e. you record events and add brief comments.

Still Photographs

Digital cameras make it quick and cheap to record scenes and photograph pieces of equipment. You may be able to use these to create a storyboard of the 'as is' and 'to be' ways of working.

You will presumably need to combine the use of still photographs with another means of documenting your observations, such as note-taking or audio recording.

Video

In human factors and user interface design work, video is widely used to analyse human-machine interactions in detail. Analysing video recordings is extremely labour intensive [3], but it enables human factors specialists to discover and solve some difficult problems.

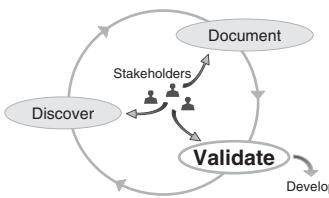
Video recording is rarely used in requirements work. This may be because it is unfamiliar in this context, but also because people feel that it will be too slow and too costly. It does have a place in observations, however, and it allows you to study rapid events repeatedly if need be.

Professional filming is extremely costly. Bernd Bruegge and his colleagues (2004) [4] have proposed *Software Cinema* in research with large manufacturing companies. This is a way of describing scenarios for future products that contain software, in a form that non-technical people can appreciate quickly.

Amateur filming can be effective. Some basic rules for video observations are:

- Plan what you want to film. Aim to record for as short a time as possible. You do not want to spend all your time editing.
- Use a tripod to keep the camera still.
- Set up a shot (e.g. of a person working at a console), focus it, and start recording. Then leave the camera alone.
- To record speech, place the microphone close to the person who will be speaking. Try to make the microphone unobtrusive in the shot.
- Make a note of the time the recording started, and the times when significant events occur in the recording.

11.4.4 Validating Observations



Once you have created a description of what you have observed, take it back to the people you observed and get them to check that it is correct.

You are very likely to make mistakes of interpretation when observing. For instance, you may overestimate the importance of some event (possibly not typical) that you have observed, or mix up some technical terms. This is nothing to be ashamed of – you are a beginning apprentice in their domain!

Make use of the fact that you are a beginner:

- ask simple questions;
- admit freely that you don't understand;
- ask for explanations.

Your questions will very likely trigger fresh comments from your stakeholders. Be ready to capture these, and perhaps to make further observations of the work.

11.5 The Bare Minimum from Individuals

Make sure you understand the viewpoints of all major stakeholders.

11.6 Exercises

1. Arrange to observe a colleague at your place of work, doing something simple like writing a business letter. Observe them as you would a project stakeholder. Record your results. Discuss what you found with your colleague. Did you choose an appropriate technique? Did you discover anything surprising?
2. You are a member of a team developing a multifunction device for sale to people enjoying active leisure pursuits in the countryside, such as

mountaineering, cycling, canoeing, orienteering, fishing, birdwatching and so on. Interview two colleagues for their requirements for such a device.

Hint: what will you want to take along to help get the interview started? What will you ask? What can you assume your interviewees will know in advance?

3. Write up your interview findings from question 2.

Hint: how are you going to organise your findings to make it easy to create requirements?

11.7 Further Reading

11.7.1 Interviewing

1. Dieste, O., Juristo, N. and Shull, F. (2008) Understanding the Customer: What Do We Know about Requirements Elicitation?, *IEEE Software* March/April, 25(2) 11–13.
An interesting and helpful article on some of the pitfalls of interviewing, with practical suggestions for doing better.
2. Jackson, M. (1995) *Software Requirements and Specifications, a lexicon of practice, principles and prejudices*, Harlow: Addison-Wesley.
Many fresh insights can be gained from Michael Jackson's book of essays.

11.7.2 Using Video

3. Jirotka, M. and Luff, P. (2006) Supporting Requirements with Video-Based Analysis, *IEEE Software* May/June, 23(3) 42–44.
A short, readable and informative article on using video for discovering requirements, written by two researchers at Oxford University.
4. Bruegge, B. *et al*, *Software Cinema*, 2004: <http://wwwbruegge.in.tum.de/publications/includes/pub/bruegge2004softwarecinema/bruegge2004softwarecinema.pdf>
Bernd Bruegge and his team have pioneered the use of film-making for software specification. Their approach would probably only suit the largest of projects, but if you are interested in getting requirements through pictures rather than words, you will find their account interesting.

11.7.3 Observation

5. Crabtree, A. (2003) *Designing Collaborative Systems, A Practical Guide to Ethnography*, London: Springer.

Books on ethnography and fieldwork tend to be heavy going. If you would like to find out a little more about the techniques and problems of tying observation to system design, this short book is designed as an introduction to the subject.

11.7.4 Tacit Knowledge

6. Polanyi, M. (1966) *The Tacit Dimension*, Gloucester, Mass: Doubleday. Polanyi's small philosophy book introduced the concept of tacit knowledge, one of the biggest practical challenges for requirements discovery. It's well worth reading the original.

11.7.5 Standard Types of Systems Analysis

7. Fowler, M. (2004) *UML Distilled*, 3rd Edition, Boston: Addison-Wesley.
8. Sommerville, I. (2006) *Software Engineering*, 8th Edition, Harlow: Addison-Wesley.

There are plenty of books on analysis, an essential background skill for discovery. Two that are up to date, accurate and helpful are Fowler's short introduction to UML, and Sommerville's popular student text.

11.7.6 Informal Modelling Techniques

9. Buzan, T. (2002) *How to Mind Map*, London: Thorsons.

There are several informal techniques that can be useful for recording interviews. One of the most successful is Tony Buzan's mind mapping. Some people love it; others do not. It is certainly a fresh and powerful way to structure a set of ideas.

11.7.7 Philosophy

10. Suzuki, S. (1970) *Zen Mind, Beginner's Mind, Informal talks on Zen meditation and practice*, New York and Tokyo: Weatherhill.

CHAPTER TWELVE

Requirements from Groups

Negotiation builds a team as well as a set of requirements.

Barry Boehm

Requirement Elements	Priorities	Measurements	Definitions	Rationale and Assumptions	Qualities and Constraints	Scenarios	Context, Interfaces, Scope	Goals	From Things	Trade-Offs	Putting it all Together
<i>Discovery Contexts</i>											
Introduction											
From Individuals											
From Groups											
From Things											
Trade-Offs											
Putting it all Together											

Answering the questions:

- What do you collectively want?
- What conflicts are there, and how can you resolve them?
- How can groups divided by distance, culture and organisational barriers work together?
 - ... so you get to hear how people's points of view fit together
 - ... so you discover potential conflicts early enough to be able to do something about them
 - ... so the team comes together and agrees the requirements.

12.1 Summary

Well-run groups work as systems: together, they can do more than all the individuals within them, benefiting from interactions. Group understanding, therefore, needs to be discovered from the whole group.

Workshops bring a group of people together, with a common aim of finding a solution to a shared problem on which they have different visions.

Requirements workshops bring together a group of people with a common requirements discovery aim, such as understanding a process, agreeing a set of goals or working out a rationale.

Obstacles to group work include geographical, cultural and language barriers.

Group media and groupware – technologies that facilitate group interaction – may help people to discover requirements despite obstacles, but they need to be carefully managed.

Group work demands both technical and social skill, especially for collaborative requirements activities such as workshops. Detailed suggestions are made in this chapter for planning and running a workshop, and for engaging workshop participants.

12.2 The Goal of Group Work

The purpose of working on requirements through groups is to get enough knowledge and energy together to uncover requirements quickly and efficiently, especially where no individuals can do that on their own.

12.2.1 Unique Capabilities

To some extent, group work overlaps with working with individuals (see Chapter 11) in its capabilities for discovering requirements. But groups can assemble information from many sources, fit it together, hear many points of

view, refine the collective understanding and reach agreement in a way that simply is not available any other way.

The traditional, and still the principal, mechanism for groupwork is for people to work together: a team sitting side by side in an office; an ad hoc group assembled around a table in a workshop or conference.

12.2.2 Obstacles

Increasingly, teams are split across two or many sites, and sometimes across boundaries of language, organisations, cultures, nations and time zones. These are serious impediments. Face-to-face meetings and co-location of teams may still be the best way of overcoming such obstacles. But these are costly, so projects have tried a wide range of approaches for group work mediated by tools, generally involving software.

The need for confidentiality (an aspect of security) is potentially a serious obstacle to group work and communication between sites, when requirements must be protected against disclosure to negative stakeholders such as competitors. Security rules may forbid electronic communication across the fence of a military site. Some negative scenarios (see Chapter 5) may relate to people inside the project or organisation, in which case they cannot be discussed openly in the group. Technologies such as encryption can help to overcome some of these obstacles.

12.2.3 Mediating Group Work (on one site or many)

Information and communications technologies – computers and networks – offer many possible ways to mediate group work, for example by sharing models or requirements, by virtual conferencing, by creating a project website or by Wiki.

Many of these ‘groupware’ tools are useful to every project, whether working on one site or on many. But how far they can be a substitute for face-to-face working depends on the quality of the ‘togetherness’ that they can provide.

This quality is often not very good, for reasons that are not just technological. For instance, working between time-zones that are five or more hours apart is simply difficult, as the number of hours available for meetings is limited. We suspect that a mixture of mediated and face-to-face group work is necessary for most projects.

In the following sections, we look in turn at:

- **workshops** – specialised requirements discovery meetings;
- **group media and groupware** – tools (software or not) that promise to mediate requirements discovery.

12.3 Workshops

A workshop is a specialised meeting, structured to bring exactly the right people together to solve a problem using a planned sequence of activities. Those people may be any stakeholders in the project, members of the development team or, if need be, external experts to assist with specific tasks.

12.3.1 Define Workshop Mission

Treat a workshop like a project: it has a mission, requirements and an implementation phase. A good workshop sets out to achieve a definite aim, and is planned and executed to achieve it.

A suitable mission for a requirements workshop can be almost anything on a project that needs the skilled involvement of stakeholders and requirements specialists, e.g.:

- to define the goals for the XYZ product;
- to create scenarios for the XYZ product;
- to draft the performance requirements for the XYZ product.

Workshops are probably the primary way to get decisions made and agreed by groups such as project teams.

Workshop activities produce requirement elements such as goals and scenarios, as described in Part I of this book. Therefore, to plan your workshop's activities, you should refer to the appropriate chapter of Part I, as well as to the element-independent advice in this chapter.

Workshops are not 'Meetings' (let alone Presentations)

Never think of a requirements workshop as just a meeting. Meetings may be called ad hoc, with no planning or structure, to address an immediate issue. They may involve presentations made by specialists, or by managers. Some business experts think ad hoc meetings and slide shows are always undesirable. But whether or not such a top-down approach works for meetings, it is quite unacceptable for requirements workshops.

Instead, we have two basic 'equations' in mind:

$$\text{Workshop} = \text{Meeting} + \text{Purpose} + \text{Planned Activities}$$

$$\text{Specific Workshop} = \text{Workshop} + \text{Element-specific Activities}$$

12.3.2 Workshop Planning

Workshops seem to work best when they are planned in detail but allowed to run quite freely. The use of time is critical, as an N-person workshop costs at least N salary-days to run, and people can quickly become bored.

Workshops as Theatre

A workshop is a kind of performance. Many people will attend, so each minute of workshop time costs many minutes of salaried time. Each participant wants to benefit from the time invested. Each participant will make a judgement on whether your workshop – and you – were worth their time.

It is therefore better to rehearse your plan in ordinary ‘slow’ time, and discover your mistakes in private, rather than to make costly mistakes in workshop time.

Workshops are like theatre, too, in that people want to enjoy the time, as well as to get results. Getting stakeholders’ ‘buy-in’ is often critical to success. An element of showmanship is necessary. That means planning, rehearsal, and playing the role.



Direct use of improvisational theatre (‘improv’) techniques is also possible (scenarios, roleplays etc); or, improv can be used as a training aid for better team-based innovation performances (see Martin Mahaux’s guest box). Simple techniques, such as acting a scenario around a flipchart model of a user interface, can be very effective.

- Budget the time carefully, and plan the workshop activities both for managing overrun (can some activities be dropped or shortened?) and underrun (should you finish early, or take along some ‘spare’ activities?). Think through and discuss the scenarios.

- Work out who to invite: all the people you need for the planned activities, and no passengers. Then, tell those people about the workshop and agree the date. This conveys the essential message that you care that the invitees attend.
- The planning of refreshments and meals is critical, as the energy of a workshop is swiftly dissipated if people melt away to do their email every time there is a break, and only return much later. Therefore, coffee, water, snacks, etc should be brought to the workshop room at planned times.
- The suggestions here are all based on experience. If you follow them carefully, define your goals for the workshop, plan what you want to happen and rehearse thoroughly, your team should be able to run a successful workshop. However, if running your own workshop seems too daunting, hire an experienced requirements workshop facilitator.

Guest Box: Mindset And Skills For The Requirements Workshop

by Martin Mahaux, inno.com

A successful requirements workshop requires careful preparation, but also a specific mindset and specific skills. Your chance of running a successful requirements workshop depends heavily on the ability of the facilitator and the participants to **innovate as a team**.

First, we need a mindset **attuned to a common objective**. The stakeholders share a common objective, but have different visions of the problem and very different sub-objectives. The common goal is to make the project a success, but the business sub-objective may be ‘more features’ and the security guys will strive for ‘more resistant to attacks’. Of course, nobody should forget their own sub-objectives, but the *credo* of a workshop participant should be:

‘My point of view is one amongst others,
and the most important is the final common objective.’

You are very likely not to achieve anything if you can’t achieve this.

At the level of your project, you can take various efficient actions to foster such positive culture. Think about it when you organise team building events; show the example when you organise team meetings; make sure people meet face to face sometimes; hook posters in the room to promote the feeling of belonging to the team; try not to segregate project roles too much. Advice on how to care about the social factors on

a project can be found in Beyer and Holtzblatt (1998) [2], Cockburn (2006) [3] and Boehm *et al* [5].

At the beginning of the workshop, ensure the facilitator recalls the desired focus on the common objective. Also, make use of techniques or methods for fostering team collaboration. Some easily applicable and efficient techniques can be found in Ellen Gottesdiener's (2002) book [1].

Once the good mindset is in place, participants need the tools to make the magic happen: the '**innovative team playing**' skills. In the toolbox are communication skills (in both directions: make yourself clearly understand, listen actively), negotiation skills, flexibility, creativity and interpersonal skills.

You know books or classical training programs won't help in providing these tools, as skills can only be learned by experience. Well, there is an unexpected place where innovative team playing experience is a discipline: **improvisational theatre** or '**improv**'.

The principle of improv is simple: a topic is chosen and, instantly, actors begin to play together a new piece around that topic. Improv sessions are like requirements workshops: the different actors enter the play with different visions and ideas about the story they will build together, with the common goal of making it a great moment for the audience, here and now, together. Improv is nothing but pure innovative team playing experience. It is also extremely enjoyable, and the discipline has developed proven training techniques to teach it step by step. So, why not use it when training requirements people? I tried it in a tailor made training program, and can't give you better advice than to try it yourself! People will learn things about themselves and about others, and will be ready to work together to build innovative requirements.

Reproduced by permission of Martin Mahaux, Inno.com

12.3.3 Workshop Rehearsal

A literal dress rehearsal is not usually necessary for requirement workshops. You are more likely to benefit from mini-rehearsals, such as:

- thinking through what you will say at each stage, and perhaps practising in front of a mirror if you are unused to speaking in public;
- trying out any group exercises such as brainstorming sessions. A rehearsal can reveal answers to concerns such as: is there, in fact, anything much to discover? How long will be needed? What resources will people need?;

- anticipating likely questions and challenges, and preparing materials to handle them;
- checking that you can be seen and heard from the back of the room.

12.3.4 Workshop Setup

The careful preparation of materials and setup is a critical part of running a successful scenario workshop.

Tested Resources

The actual resources you will use must be instantly available, and tested on the spot so you know they work.

- If you are using electronic equipment like computers, projectors or electronic whiteboards, test it, and ensure you have spare equipment to hand.
- Workshops do not have to be run with any electronic equipment; they can work better without, as the human contact is then more natural. Writing and drawing on flipcharts can be fast and effective.
- Check the supply of paper and pens; flipchart pens usually contain permanent ink, which is a small disaster on whiteboards.
- If you mean to draw on whiteboards (necessary if you need to make large drawings), take a digital camera (with spare batteries and plenty of free memory) to record the results.
- If participants are to be provided with reading materials such as background notes, make sure these are printed and bound in good time. Ensure they arrive at the venue safely: it is no good if they get lost in the ‘goods inwards’ department on the day of the workshop.
- If you are using a venue like a hotel, and you depend on them to photocopy materials for you quickly during the workshop, test them out first to ensure they really can respond reliably. Venues sometimes advertise photocopy and fax facilities when all they have is a slow multipurpose machine at reception, which reception staff are too busy to operate.

Room Setup

- The workshop room must be set up in advance.
- The traditional room layouts of a circle or U-shape work well, as they imply equal status for all participants.

- If possible, set the room up the day before; failing that, at least look at the room, work out how to set it up and arrive early to shift tables, etc.
- Aim to give everyone the feeling that you know exactly what you are doing, that the workshop is well planned and that their participation is important.

Fix Problems in the Room

Ensure that you discover and fix any problems in the room. Common ones are listed in Table 12.1.

Setting 'House Rules'

- There is usually one person in a group who likes to talk continually, so plan what to do to prevent that. For example, state the 'house rules' at the beginning, emphasising that time is limited, and you reserve the right to park issues for discussion after the workshop.
- Minimise other disturbances:
 - ask people to switch off their mobile phones, etc;
 - stress that you require everyone's full attention, as their silence will be taken as complete agreement to all decisions taken during the workshop.
- Set roles for people:
 - **Time-keeper** to announce 'coffee time', etc. This frees the facilitator to focus on discovering requirements but, more importantly, it says that the workshop is a communal effort, not a performance by one or two people.
 - **Rat-hole watcher** (as suggested by Beyer and Holtzblatt (1998) [2]) to announce 'this is a rat hole' when the discussion is wandering way off the subject (down an infinitely-branched and well-worn but irrelevant hole). This implicitly gives everyone, not only the official hole-watcher, permission to look out for rat holes.
 - **Dictionary owner, issues list owner, data modeller**, etc. These people can share the work of recording and organising workshop outputs. But again, the real purpose of doing this is deeper. It is to alert people to the fact that arguments over terms, or anxieties over project issues, can be raised consciously as things to be addressed (rather than traps to fall into).

Table 12.1: Solutions for problems in workshop rooms.

Problem in the room	Solution
Poor lighting (e.g. inadequate blackout; lamps pointing straight at the projection screen).	<ul style="list-style-type: none"> Visit the room. Try out the lighting. Make sure you can read what is projected from the back and sides of the room. Ensure that all the issues are addressed before the workshop.
Inaccessible power supplies.	<ul style="list-style-type: none"> Bring extension cables and gaffer tape. Ensure the cables cannot cause people (including you) to trip. Tape them down well.
Broken electronic equipment such as projectors; Incompatible equipment; Video and network cables with bent connector pins; Network sockets switched off, etc.	<ul style="list-style-type: none"> Test everything as if you were holding the workshop. Assume Murphy's Law applies ('If anything can go wrong, it will').
Excessive amounts of furniture.	<ul style="list-style-type: none"> Ensure the room only has what you need, with the maximum of space to move around. Stacked chairs and tables give an immediate impression of clutter and chaos – exactly what you don't want in a workshop.
Poor ventilation, heating or cooling.	<ul style="list-style-type: none"> Find out how to open windows and work the air conditioning in advance. Ensure there is water for everyone to drink. If the air conditioning is making participants uncomfortable, turn it off. Remember that seated participants can become much colder than active facilitators, especially if they are sitting right under an air conditioning duct.
Inadequate wall space, or forbidden to pin/stick flipcharts to the walls	<ul style="list-style-type: none"> Find a room with blank walls, glass partitions or other surfaces you can stick things on to. Technological fixes include: hanging backing-paper (e.g. newsprint) to stick things on to; electrostatically-charged folio flipcharts that stick reversibly to any surface; temporary screens or pinboards.

Guest Box: How to Engage Participants in a Workshop

by Joy Beatty, Seilevel

Workshop sessions should be kept as short as possible, ideally one to two hours. However, some projects will require longer sessions, sometimes for multiple days in a row. Day-long workshops are at high risk for being tiring, boring, and uninspiring. To combat this, use activities that will keep participants mentally alert, while encouraging them to actively think about the topics and share ideas.

The following suggestions are specific examples to help engage workshop participants.

Move Around

Participants will stay more attentive in workshops if there is active movement in the room, and it is best if they are part of the movement. They can take turns writing notes on the wall, to keep them alert and give them ownership over the content. The official facilitator and scribe will still be responsible for making sure the substitutes are doing a quality job, with subtle guidance as needed. The person at the wall should *not* be the lead expert on the specific topic being discussed.

When the energy dropped in a process flow workshop for engineers, the scribe at the wall tossed the whiteboard marker to another engineer to take over. Trading-off the scribe role continued throughout the session. The participants enjoyed the opportunity to participate in some way other than sitting down.

Play Games

Games encourage cooperation and friendly competition, which help engage participants and inspire insights. Games should be used to produce valuable content without slowing the workshop down significantly. The game rules must be simple. It helps to have small prizes for participants who offer particularly useful insights or active participation, so that the emphasis is not just on winning.

A game was used to identify features using affinity diagrams, with a twist that required silence (based on Thiagi's *Silent Movies*¹). The workshop attendees had blank sticky notes, on which they listed individual features that came to mind. Then, without talking, in small groups they categorised the collective set of features on a piece of paper on the wall.

¹Thiagarajan, S. (2003) *Design Your Own Games and Activities: Thiagi's Templates for Performance Improvement*, San Francisco, CA: Pfeiffer

If the group members disagreed on features or categories, they had to argue *silently* to come to resolution. In the end, the facilitator brought all the papers together, leading verbal discussions focused on the quickly identified differences.

Use Toys

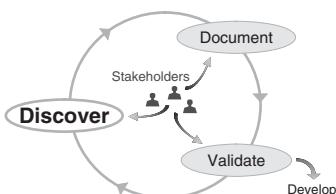
Placing toys on the workshop tables supports a fun and relaxing environment, while alleviating fidgeting and fostering creativity. It is important that the toys are kept simple, so as not to be too distracting. It is best not to pick mentally challenging puzzles that require concentrated attention to play.

For example, Play-Doh® (a modelling paste) was given to a group of executives who were brainstorming product innovation ideas. They were asked to model with Play-Doh what ‘product innovation’ meant to them each personally. The models they created included a dollar sign, a box with a ball that did not fit in the box, Yin and Yang, and a window to the world. The activity revived the creative minds in these executives after a number of hours of concentrated thinking.

As a final suggestion, activity selection should be based on the type of attendee, the goals, the context, and the schedule of the workshop, so as to create an effective and engaging experience.

Reproduced by permission of Joy Beatty, Seilevel Inc.

Workshop Activities



Workshops can provide the context for discovering (or starting to discover) many of the types of requirement described in Part I of this book.

For instance, your project might need to use one or more of these activities:

- **Stakeholder onion/influences model (Chapter 2):**
 - Draw a diagram of the stakeholders involved, and the influences they have on each other.

- **Brainstorming and clustering goals (Chapter 3):**

- Ask participants to write their goals on sticky notes in, say, 5-10 minutes.
- Ask participants to stick their notes on the wall.
- As a group, cluster the goals into groups on the wall, and give each group a title.
- Have everyone sit down, and take feedback on the activity.
- Give everyone a second chance to write down any missed goals.
- Cluster those goals also.

Emerging Issues

Do not be disappointed if a workshop output, such as a cluster of sticky notes, is not what you would call a goal. If a workshop is more concerned with issues like lack of training, or the difficulty of working across multiple sites, then that is what they immediately want to deal with. Of course, the *project* goal ‘provide training’ is the flip side of the issue. There is no reason why all the goals should be for the *product*.

Once you have respected and dealt with people’s concerns, goals will emerge without effort.

- **Goal model (Chapter 3):**

- Brainstorm the goals of different stakeholders.
- Draw a goal model to identify synergies and conflicts.

- **Threats, risks, obstacles, and countermeasures (Section 3.2.6, The Negative Side, in Chapter 3):**

- Identify negative aspects by brainstorming, war-gaming ('I could defeat that by doing XYZ'), walking through scenarios or scanning a template of known risks.
- Identify countermeasures to each negative item.

- **Context model/rich picture (Chapter 4):**

- Draw whichever kind of context diagram is most appropriate, showing what lies just outside (and if need be, what lies just inside) the boundaries of the system. Explore the possibilities by moving the boundary to include more or fewer items.

- **Scenarios (Chapter 5):**

- List scenarios on flipcharts or whiteboards, and be ready to correct them as you find out more. There are several useful workshop techniques for scenarios; scenarios and workshops go well together, forming a quick and powerful means of discovering and agreeing requirements with stakeholders.
- For detailed suggestions for scenario workshop activities, see Chapter 6.
- Ellen Gottesdiener's (2002) *Requirements by Collaboration* [1] makes many practical suggestions specifically for scenario workshops.

- **Qualities and constraints (Chapter 6):**

- Workshop with business users, scanning a template of typical NFRs.
- Work in specialist areas such as security or safety; for instance, a safety team could run its own workshop.
- Run a show-stopping constraints workshop, e.g. to bring together people from different areas to identify the top risks to the project.
- If not already done, run a goals workshop to agree the key qualities of a product.

- **Rationale and assumptions (Chapter 7):**

- List the major assumptions that participants are making in their scenarios, or in other documents.
- Create rationale diagrams to explain the reasoning behind any critical decisions facing the project, or to capture any arguments that arise, in a neutral way.

- **Terms/definitions (Chapter 8):**

- Once you have already held a scenarios or other workshop, you may find you need to run a workshop activity (probably not often a whole workshop) to resolve terminology issues.
- Make a list of project dictionary terms, and get participants to define each term. If there are competing definitions of some terms, invent new terms to go with each definition. (Also watch out for synonyms, etc.) Disagreements on terms can be very revealing; resolving them can quickly lead to progress.

- **Priorities (Chapter 10):**

- Debate, vote, sort cards, cluster sticky notes, etc, to obtain agreement on the importance of the requirements to stakeholders.

- **Trade-offs (Chapter 14):**

- Brainstorm possible design options.

- Score the design options, once these have been worked out in sufficient detail, on the criteria (based on requirements).

Tips for Workshops

- Plan carefully.
- Test anything that seems risky or uncertain in advance of the workshop.
- Use techniques that you know will work for you – live workshops are not the right place to experiment.
- Visit the venue and do a ‘dry run’ as if you were running the workshop.
- Talk through the schedule with the venue manager – make sure you agree what is going to happen, and when.

Combination Workshops

The most basic advice for workshops is ‘do one thing at a time’.

But it can be right to work through several activities, involving more than one requirement element (from Part I of this book) for a single workshop.

For example, one successful workshop went like this:

1. ‘**Theatre**’ sketch setting the scene (a few minutes).
2. **Scenarios:** recorded from each of three areas of work involving apparently the same general kind of problem, but which were rightly suspected of having strongly differing equipment needs (most of a day).
3. **Brainstorming of goals** derived from all participants’ reflections on the scenarios they had just witnessed (half an hour).

Complex Topics

By their nature, complex topics are all different. Often, you can split a complex topic into a series of simpler questions, such as:

- What are the stakeholders’ goals for this? (Chapter 3)
- What conflicts have arisen? (Chapter 3)
- How do we see this working? (Scenarios, Chapter 5)
- What are the arguments on each side? (Chapter 7)
- Can we imagine different resolutions to the problem?
- What are our priorities here? (Chapter 10)

Such questions will often match a chapter in Part I of this book; and they form likely candidates for separate workshops – or at least, major activities within a workshop.

Perhaps the most common complex question that faces projects is that of selecting an approach that best meets a set of goals, i.e. trade-offs (Chapter 14). That is a matter of both requirements and design, so a resolution will usually involve different teams, often from different organisations. Trade-offs are good subjects for workshops, but these demand very thorough preparation, including the design of different options (not just the preparation of the workshop as a meeting).

Large Groups

Large projects usually mean complex problems and large teams: a poor mixture for a workshop. If possible, meet people individually first, and hear their points of view.

Age-old wisdom is that effective teams and meetings are quite small. A general, for instance, commands a huge army, but his planning meetings involve only a handful of people, who all know each other well. Parliaments consist of hundreds of delegates, but government is by a cabinet of ministers who can all sit around a table together. You can't have a shared decision-making workshop that fills a lecture theatre.

It is therefore critical to find a way of bringing group size down to an effective number. Even 20 is becoming large for a workshop. After all, with 20 people in a room, each person can speak for three minutes per hour on average; and many people will not speak at all. Facilitating a workshop of 30 people is possible, but only just. As you add more participants, the meeting fatally changes its character from a participative event to a lecture or performance.

We have attended, and occasionally been forced to run, supersize workshops of various kinds. The organisers try various tricks to give people a chance to feel as if they are getting involved: splitting into breakout groups, each with a topic to examine; appointing a spokesperson to report on the group's findings; preparing one or two slides of summary; presenting the slides in a rushed, two-minute slot. The overall result is a mass of detailed organisation on the day; a lot of 'Now we will call you back into the main room in 10 minutes', plenty of shuffling about and rather inconclusive results.

So, workshops must be small. Small teams should run separate workshops. Findings can be reported to other teams. Team leaders can get together. Interfaces between two teams can be agreed locally. These technical activities should be quite separate from any desire to get everyone together for other purposes.

Not to be Attempted

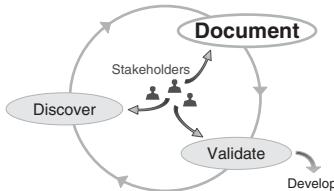
It is hard to legislate on what not to do in a workshop; some of the best workshops achieve success with agendas that look impossibly ambitious.

There is value, too, in a degree of hustle; time flies in the best workshops.

But, in general, do not try to do too much. Having a single goal, with several related activities, is more likely to work than a hybrid mishmash of a workshop that attempts to do everything.

We have emphasised workshop planning. But workshops can succeed under pressure, and may thrive on it; planning can be done quickly, as long as good thinking goes into it.

12.3.5 Workshop Recording



The task of recording a workshop's findings is important. An accurate record, shared promptly with all participants, builds trust that the process is fair.

A full-day workshop (for instance) can generate a large volume of findings. You have two conflicting goals:

- to record every finding;
- to allow the workshop to proceed smoothly (at full speed).

Drawing by Hand

You can draw diagrams by hand and record them with a digital camera (if allowed), on a self-printing whiteboard or simply by keeping flipchart sheets. This is probably the best approach for a workshop. Drawing with pens is often faster and less fiddly than using software. It is also more reliable. But the main advantage is that it supports human communication better. For example, participants can see that you draw and label their bubble when they call out a goal. They understand that they are being heard. This feels quite different from presenting people with a clever model that you dreamed up. You can quickly correct mistakes on a whiteboard, too.

Your solution will vary, but it will involve:

- enough effort to document results as they appear (e.g. **scribe** with computer, **facilitator** with flipchart pen, etc);
- appropriate choice of recording tools and techniques (flipcharts, whiteboards, writing materials for participants, camera, voice recorder, etc);
- a suitable set of activities for participants, some of which will involve them writing things down, drawing sketches, acting scenes, etc.

Flipcharts have the great advantage that they can be put up on the walls of the workshop room, offering immediate feedback that the workshop is being productive, and participants are being heard.

Guest Box: A Three-Person Facilitation Team

by Dr Frank Houdek, Daimler

On a recent workshop, I observed a novel workshop approach with a larger than usual team of facilitators.

The **moderator** documented everything on flipcharts. The flipcharts were put up on the walls so they were visible to participants all the time.

A **scribe** from the client organisation recorded the findings on a laptop.

An **observer** (from the same consultancy company as the moderator) sat beside the scribe and helped her. His major role was to act as quality control. If he got the feeling that the moderator had got something wrong from the participants' statements, he joined in discreetly. He asked questions like: 'Maybe I got it wrong, but . . . ', leaving the moderator in the 'active position'.

Reproduced by permission of Dr Frank Houdek

Recording a multi-activity workshop, especially if there are parallel streams, is complex. Plan and rehearse how you will record and collate all the findings. For example, you could decide to label each flipchart sheet with the date and activity that it records.

Recording with a Digital Camera

A digital camera is a huge help in recording workshop findings.

Common problems include:

- glare from the reflection of the flash on whiteboards, affecting readability;

- poor quality images (odd colour, low contrast, distortion, fuzziness);
- inconveniently large image files.

A few tips will help you get much better results:

- Stand slightly to one side to avoid reflections.
- Shoot a test photograph; if the flash causes problems, turn it off.
- Process images to give them a standard, good quality appearance.

Image processing need not be complicated. Work out a simple approach that does the job for you. You will need a ‘paint shop’ style software package (one is often provided free with digital cameras). A basic sequence is:

1. **Perspective correction** (to make whiteboards appear as undistorted rectangles). Typically, the tool shows a rectangle in front of your image. You stretch its corners to match the corners of your trapezium-shaped whiteboard, and double click to apply the correction.
2. **Colour balance** (use the ‘fluorescent bulb’ or 4200 Kelvin setting for room lighting).
3. **Contrast enhancement**.
4. **Unsharp mask**.
5. **Crop** (to remove blank areas around the edges).
6. **Save as** a JPEG image with medium compression.
7. If the file size is still very large, save a copy of the image, and **resize** the copy to (say) 30% of the original. Sharpen it and check it is still legible.

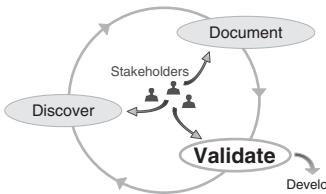
Many of these steps may be available on your package in a command named something like '**One Step Photo Fix**'. Better, your package will probably be able to record a sequence of commands as a script that you can re-run with a single command.

Make an index or overview document to help people find their way around the JPEG images. This can be a Word document, or a simple web page with named links to the images.

Document files may become very large if you drag images into them. Word files stay much smaller if you use Word’s **Insert, Picture, From File ...** command. It helps to keep groups happy if you send them small files.

Finally, there is a trade-off between sharing images quickly and taking a little time to translate the images into proper models. Don’t spend hours on image processing.

12.3.6 Validating Workshop Findings



A good workshop should leave you with the feeling that you understand what the stakeholders want, along with a detailed (if handwritten) record of the workshop's findings. This is a human thing, a warm, shared assurance based on the trust developed in the workshop (see the guest box on 'Building and Sustaining Trust', by Ellen Gottesdiener).

Therefore, validation should mainly consist of checking that you have not lost anything when tidying up the findings afterwards. This is most simply done by asking the participants to check your workshop report.

Guest Box: Building and Sustaining Trust

by Ellen Gottesdiener, EBG Consulting

A fundamental ingredient of successful requirements workshops is building and sustaining trust. This may seem 'touchy-feely' and vague, yet there are mechanics for trust building. I find them consistently useful in the workshops I have planned, designed and facilitated, and I use them in every workshop.

Some of these techniques are necessary to employ before, during, or after the workshop event. I associate them with three types of trust: contractual, communication and competency.

Building trust before the workshop

Form a planning team composed of a balance of perspectives (business and technical) that will assist the facilitator in planning and design. Craft a clear purpose that resonates with all workshop stakeholders. From there, define the workshop products or outcomes. Most of your workshop products will be requirements in some form, but your outputs will also include decisions (e.g. what is in and out of scope, what the business rules are for each set of scenarios, the priority of requirements, what stories to deliver in a given iteration, and so on).

Let's face it: requirements development is a series of ever-unfolding decisions. Prior to the workshop itself, communicate to all stakeholders what the decisions need to be, what decision rules the group will use,

and the decision-making processes the facilitator will employ to help the group reach closure.

Defining your purpose and products (including decisions) builds **contractual trust** – that is, trust based on clear expectations, mutual interdependency and joint accountability. Contractual trust is built by delineating workshop roles and responsibilities early on (e.g. the facilitator, recorder, participants, sponsor, observers and support cast).

Building **competency** is another ingredient for trust: the ability of people to rely on each other, seek input, and obtain feedback from each other. This means getting the right participants, including a mix of subject matter experts, customers, and technical staff – the people who know the problem domain; those who will use the product; and those who will build, supply, support and maintain the solution.

Another type of trust, **communication trust**, involves a group's ability to enjoy open, honest, and frequent communications (truth telling, with good purpose). Set the stage for communications trust by actively seeking out hidden agendas, so you can prepare appropriate activities or obtain permission to surface difficult issues or 'undiscussable' topics during the workshop.

Building and sustaining trust in the workshop itself

Start by having the sponsor present and briefly 'kicking off' the workshop. She or he should state why the workshop is important, expectations of the group's work, and how they will support those outcomes. This adds to contractual trust.

After the kick off, and as part of setting the stage for getting down to work, workshop participants should establish working agreements or ground rules for how they will conduct themselves during the session to building contractual and communications trust. Include decision-making rules, which in trustful groups is transparent and explicit.

One of the facilitator's greatest challenges is facilitating in the face of conflict. Welcome conflict! It's a normal part of group dynamics, and its absence may indicate suppressed issues likely to surface during the workshop. Conflict is also an indicator of energy to be harnessed for productive ends. Workshop participants build communications trust when they directly and honestly discuss conflict. By using agreed methods for making decisions to achieve closure, and openly addressing misunderstanding and disagreement, the facilitator helps the group manage conflict and boost communications trust.

Effective requirements workshops rely on participants' interdependent knowledge and skills. Playing with multiple interwoven analysis models (e.g. scenarios or stories, business rules, data model, actors and so on)

builds the group's competency with the problem domain, while concurrently maintaining effective communication. Requirements workshops require continual learning and feedback. When participants 'play' with multiple requirements models, they push the edges of their knowledge and skills, and learn from each other. Multi-modeling verifies requirements. At the same time, participants test each others' inferences and assumptions about those requirements. This builds both communications and competency trust.

Sustaining trust post-workshop

I recommend conducting a sponsor or key stakeholder 'show and tell' at the end of the workshop or series of workshops to increase communication trust up and down the organisation. These 'show and tell' events (which can be conducted a day or two after a workshop, if necessary) increases management's competency with the requirements. Since management folks often don't participate in detailed requirements workshops, yet need to understand the impact, risks and trade-offs associated with them, hearing outcomes in a 'show and tell' increases their knowledge so they can better support the project.

Be sure to share the workshop documentation with all participants as well as the larger stakeholder community soon after the session, thereby sustaining communication trust.

Your final workshop activity should be a workshop retrospective (i.e. debrief, self-reflection, lessons learned) that takes a hard look at how the workshop went, what worked well, what could be improved, and actions for improvement. The retrospective explores both product (workshop deliverables) and process (effectiveness of group interactions and workshop planning and communications). Retrospectives build on all three types of trust – contractual, communication and competency.

I also recommend conducting a post-workshop retrospective two to four weeks after the workshop or series of workshops. Now that some time has gone by, do we still think that the workshop was a good use of our time? What did we do really well? What deliverables from the workshop are we using now to build, test, and deploy the product? Did we get the right ones during the workshop? In retrospect, do we have the right people?

Reflecting on the utility of the workshops helps you build overall organisational competency with requirements workshops. After all, we want our workshops to be positive and productive, and truly achieve an experience in which, as Aristotle remarked, 'the whole is more than the sum of its parts'.

The output from a workshop often involves various models from Part I of this book (e.g. goal models, scenarios, rationale models). You should check those using the techniques listed in the relevant chapters of Part I.

12.4 Group Media

Documentation is the worst form of communication.

Alistair Cockburn

Group media include any sort of tool, usually software, that facilitates group work. Categories of special interest to development projects include:

- project wall (a real wall, not software);
- project website;
- project Wiki;
- modelling tool;
- requirements management tool;
- groupware (software designed specifically to help groups to work).

We should say at once that none of these, not even the next-to-last, is designed explicitly for discovering requirements.

Other tools that can be called groupware include software for negotiation, voting, and issue tracking, as well as communications tools, including anything from email and instant messaging through to videoconferencing.

Let us consider the advantages and disadvantages of each in turn.

12.4.1 Project Wall

Teams traditionally work in a single physical area, for the good reason that this is effective: it facilitates communication and makes management easier.

A simple innovation suggested by the ‘agile’ school of software development is to provide a conveniently located wall, such as a glass partition, on which everyone in the project can post shared information.

For

At one extreme, a wall can simply display ‘top-down’ project notices. But a wall encourages a more democratic approach. People can share work-in-progress, such as a large sketch of a current software architecture, or lists of risks, assumptions, issues to be resolved, questions that someone else might be able to answer, and so on.

When a team works in a single area, a project wall can be an excellent choice of groupware. It is on show all the time; it is informal, inviting contributions; it is extremely cheap; and it requires no special training course.

The choice of wall is important. It needs to be in a place where people regularly go – near the office kitchen is often good – and it must be large enough to invite contributions.

The wall can form a good backdrop for team meetings. For instance, a team in one automotive company has an effective weekly stand-up meeting in front of current defect data charts to discuss progress.

Against

A wall is purely passive; its use for discovery depends heavily on the enthusiasm of team members. A wall is not good at displaying large documents or for permanent records (e.g. of published versions of requirements).

People can probably only be expected to pay attention to a few 'hot' issues at a time, and responses may arrive over a period of days.

A wall can't provide the focus and energy of, say, a workshop.

Remote teams can't see the wall.

Walls can be used well or badly, seriously or frivolously. They aren't guaranteed to work on every project.

12.4.2 Project Website

Many projects now create their own intranet website (Figure 12.1).

For

A project website is invaluable for sharing standard procedures, dates of meetings and deadlines, document templates, management decisions, training materials and so on.

It is also (unlike a project wall) an ideal place for publishing baselined versions of project documents, including requirements, for all to refer to.

Against

Despite all the advances in website editing, a website can still feel like a 'read-only' noticeboard. In practice, only a few people on a project tend to update the website, or perhaps one person is given the role of collating and publishing project information. This makes a project website rather a passive medium and, as such, not very effective at stimulating the discovery of requirements.

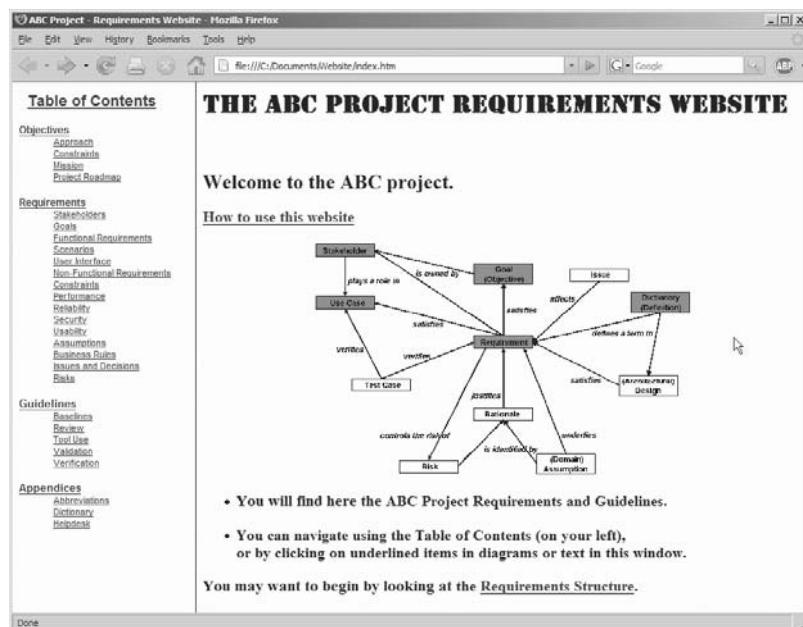


Figure 12.1: A project website.

12.4.3 Project Wiki

A Wiki² is a tool available on a network via an ordinary web browser. Its user interface is a cross between a hypertext and a text editor. Wikis also often keep a history of changes, so people can see who created what, and can revert to an earlier text if need be.

For

A Wiki allows a team to create a shared structure of linked pages, which could describe a project's requirements, design, issues, day-to-day decisions, and so on.

Wikis seem to work best when there are 'topic owners' who actively keep charge of discussion on specific topics.

Some Wikis allow group members to 'watch' for changes to an item, such as the usability requirements page. Anyone may change the page (you may wonder if that is appropriate, if the page depends on specialised knowledge)

²Wiki means 'quick' in Hawaiian. Wikis were invented by Ward Cunningham in 1994. Perhaps the largest Wiki, and certainly one of the best known is Wikipedia (http://en.wikipedia.org/wiki/Main_Page), an encyclopedia edited by the web-using public. It currently contains over 9 million articles written in over 250 languages.

but the usability team are then immediately notified of the change by email. This encourages debate, which may be useful for difficult requirements.

Against

The ease of editing a Wiki gets over some of the weaknesses of the project website as groupware, but also creates challenges of its own:

Wikis sometimes stir up controversy, rather than facilitating agreement.

Wiki pages can be of variable quality, and may quickly be forgotten.

The very ease of editing can encourage team members to keep on ‘improving’ the requirements, causing scope creep (see Chapter 4). This is uncontrolled ‘discovery’.

If not managed by someone, a Wiki’s lack of framework can lead to chaos.

The structure of a hypertext is, in a way, very suitable for requirements, as you can create a hierarchy (e.g. qualities include usability and dependability; dependability includes safety, security, and availability, etc) with cross-references for dependencies, and so on. However, this can make it hard to extract a simple, printable document (e.g. a list of all the requirements).

There may be little support for basic project essentials like making a read-only (baseline) copy of the requirements, or recording attributes of requirements (like status and priority).

Current Wikis, therefore, are promising as requirements discovery groupware today, but they need to be used with some care. Perhaps advances in Wiki tools³ will make them more widely suitable for projects.

12.4.4 Modelling Tool

Specific types of modelling tool are described in Appendix C.

The larger modelling tools provide varying degrees of support for group working, starting with the basic ability to share models: the ability to lock individual diagrams, and to notify other users who are currently working on a model. Modelling tools may also offer more Wiki-like facilities, such as history and discussion; or more database-like facilities, such as versioning and linking to requirements.

³For a discussion of the advantages and limitations of Wikis, see: O’Leary, D.E. (2008) Wikis: ‘From Each According To His Knowledge’, *Computer* 41 (2) 34-41

For

The special thing about a modelling tool as discovery groupware is the model itself, presented to the whole group. Almost any kind of model can stimulate requirements discovery if presented in the right way to the right people. A modelling tool that enables your team to build, discuss, and edit requirements models together should, in theory, be invaluable.

Against

The practice may be rather different. Tools have limitations; they are costly, and take time to set up and to learn. Mainstream tool providers have mostly not given much attention to requirements discovery; there is excellent support for flowcharts and UML class diagrams, but very little for goal and rationale models, for instance. You may have to do awkward things (like putting requirements into comment boxes, or modelling assumptions as classes) to get around tool limitations.

12.4.5 Requirements Management Tool

The use of requirements management (RM) tools for discovery is described in Appendix C. Our question here is how effective they are as discovery *groupware*.

For

As with modelling tools, RM tools offer facilities for sharing requirements, including locking, passwords, history and versioning.

RM tools inherit the benefits of databases – linking information of different kinds, searching, filtering, tracking status, and so on. All of these are valuable in enabling groups to discover requirements, as they facilitate checking for completeness, correctness and consistency.

Now that RM tools are adding the ability to edit models of various kinds, including scenario storyboards, for instance, they are becoming better at supporting discovery directly (and they are turning from 'pure' RM tools into project information and project life cycle management tools).

Against

RM tools are primarily designed to manage rather than to discover; and to manage text rather than more direct forms of communication.

Like modelling tools, they can be costly and time consuming.

RM tools can feel unwelcoming to people who do not like software and computers. There can be an air of heavily imposed 'process', offering team members little stimulation to think creatively.

12.4.6 Groupware and Working at a Distance

There is a certain magic that happens when full-time, dedicated project members occupy the same space.

Tom DeMarco *et al* [4]

Having to work at a distance, sometimes across multiple barriers – organisational, linguistic, and cultural, not to mention time zones – is a major challenge for development projects. Distance creates many risks for projects, including a commonly experienced mixture of errors, delays, misunderstandings and rework (Table 12.2).

Numerous 'groupware' tools designed to help with working at a distance have been marketed, using many different technologies including video/web conferencing and telephone conferencing.

Well known examples of groupware include IBM Lotus Sametime, Cisco WebEx, Pixion Picture Talk and Microsoft Live Meeting, among many others.

Such tools offer alternatives to costly and inconvenient travel for actual face-to-face meetings. Some of the tools are themselves costly, though advances in telecommunications and especially the Internet have brought cheaper options.

Table 12.2: Risks of splitting development teams.

Type of barrier	Risks to project	Mitigations
Organisational	Incompatible procedures, documentation styles, notations, resource management. Hence, context not understood, requirements misinterpreted, key staff not available.	Discuss and share procedures and templates; attend training courses together; have some team members co-located; have team leaders visit other team(s) regularly; share a meal, sport, cultural activities; explain requirements and issues visually, not just in text.
National culture	Context not understood, requirements misinterpreted. Some cultures avoid saying anything that might cause someone to lose face.	
Distance, time zone	Meetings become rare/awkward; issues get aired only after long delays; problems build up.	

Collaboration at a Distance



From foggy London to the sunshine of India

In the last few years, organisations have tried to save money on software development by outsourcing. For example, banks have written software specifications in Europe, for coding by the booming software industry in India.

Last year, an English retail company decided to follow suit, passing its requirements to an Indian software house. The software team, experienced in financial software such as banking and insurance, worked hard to meet the tight deadlines, but had difficulty understanding the requirements in the new domain. There was no one to discuss them with. Work was delayed by other unavoidable local factors too, including the monsoon rains.

Meanwhile, the executives of the English company were growing impatient. Why weren't people working faster? They were unaware of the local issues, and could not see how hard the software team was working. Having the 'bandwidth' of communication reduced to written specifications, emails and a few phone calls was seriously limiting progress.

The two organisations realised they both needed to discover which aspects of their different cultures and assumptions the other was having difficulty with. This need was met by having members of the team travel to the other country, taking time to experience the issues. The project never became as straightforward as a local one, but the effort did help to communicate the real requirements. Perhaps paper alone is never enough.

These tools may have started life as attempts to substitute for unmediated group work, which is in some ways the ideal. However, groupware tools now offer useful capabilities that can be better than a traditional face-to-face meeting for some purposes.

Capabilities of groupware include:

- instant messaging ('chat' by typing short messages);
- point-to-point video conferencing ('talking heads' of other participants – two-way or multi-way);
- multi-way voice calls;
- slide shows with voiceover (used in 'webinars');
- virtual whiteboard (shared editing of diagrams and sketches);
- shared desktop (you show participants spreadsheets, etc, while you talk about what you are showing);
- distribution of files (as if you were giving out handouts at a meeting).

New features are constantly being added to groupware tools.

- Some features of groupware tools, such as video conferencing, simply help distributed groups to work almost as well as if they were in one place together. These features do not contribute specifically to discovery: they simply reduce the negative impact of fragmentation on projects. Discovery, however, is a task that is seriously disrupted by distance, as it critically depends on close and continued collaboration. The other major role of requirements, communication of needs, is also very vulnerable to distance (see the box, 'Collaboration at a Distance').
- Other features actively bring people's ideas and contributions together, for example, negotiation and voting (e.g. with Barry Boehm's WinWin approach [5]) and the virtual whiteboard. These can facilitate discovery, and consequently are also useful for groups working on a single site.

Relying on groupware to overcome these obstacles must be seen as a risky approach. You should understand, for instance, that web meeting or video-conferencing facilities will not in themselves overcome the barriers listed in Table 12.2.

12.4.7 The Role of Group Media

Group media, including groupware, have a definite role on the typical project (see Table 12.3, which summarises the discussion in this chapter). Whether their role on your project is essentially as a notice board, or whether you should go further and try to stimulate and manage discovery with a tool, is a decision for you.

Table 12.3: Summary of pros and cons of groupware approaches.

	Groupwork approach	+	-	Points to note
Face-to-Face	Workshop	Power of dialogue and human contact to solve problems, reach agreement	Cost (many salaries, time to plan, need for travel); can be overused	Workshops can create a unique, high-energy environment able to overcome obstacles
Mediated group work	Wall	Shared: democratic, low-cost, informal	Can be ignored; no good for large documents; best if whole team is in one office daily; can be misused	Not suitable for teams split across different sites, or even different buildings on one site
	Website	Available on every PC; ideal for sharing procedures, templates, decisions, baselined documents, etc	Often has read-only feeling, so can be too passive for discovering requirements	Depends on culture; not everyone likes to use software tools as communications medium
	Wiki	Democratic; encourages fine-tuning of text; keeps history	May open debate but fail to close it; can cause endless editing; may make printing etc difficult	
	Modelling tool	Models visible and editable directly; displays concepts and opens them to challenge	Cost and time to set up and learn tool; not many tools help much with requirements	
	RM tool	Many features to help with documenting and managing requirements	Less help for discovery in many RM tools; strong emphasis on text; cost and time to set up and learn	
	Groupware	Attractive to use technology to try to cope with working at a distance	Groupware can simulate proximity, but research shows it is not as good as the real thing	Video conferencing may not overcome barriers of culture and distance

Crossing Cultural Barriers

Engineers in a German company were collaborating on a project with colleagues in China. In Europe, they used an electronic ticketing systems to document and track software bugs.

The Chinese colleagues had real problems with such a system, as using it would mean blaming a colleague, so ‘face’ would be lost. No amount of persuasion from the European side had any effect.

In the end, the Chinese engineers agreed to assign tickets to European colleagues (who were not part of the project). The Europeans then reassigned the tickets to Chinese co-workers. Getting a ticket assigned from Europe was acceptable, as the strict unwritten rules of ‘face’ are not assumed to apply in that case.

Group media are not a substitute for face-to-face meetings and for a team simply working closely together.

Some people really don’t like having to use software tools, either. Choose tools and techniques with your team. For instance, if no one in the team likes creating Wikis, choose a different approach that they are happy with.

As software evolves, and as teams more often find they have to work at a distance, groupware will probably become increasingly attractive.

12.5 The Bare Minimum from Groups

- Agree any scenarios and issues that affect several stakeholders.
- Make sure the ‘bandwidth’ of human communication is enough to enable issues to be shared and resolved promptly. That ‘bandwidth’ is not the same as the promised capabilities of groupware tools.

12.6 Next Steps

Prioritise the requirements, if you have not already done so (Chapter 10).

12.7 Exercise

You are on a multinational project based in California, USA; Sydney, Australia; Bangalore, India; and Rotterdam, The Netherlands. The project is six

months into a three-year development, and is already three months behind schedule. Communication between the teams is proving difficult. The team in Sydney is becoming unhappy with teleconferences at inconvenient times of day, and they feel they are not being listened to. The project manager in California feels that the other three teams are not pulling their weight. The technical leads in Bangalore and Rotterdam are trying to get California to agree the main interfaces they have proposed between them, so they can start on design proper, but feel that California is insisting on making all the decisions.

The project manager has asked you to fix the communications problem between the teams.

- a. What kinds of groupware (software or otherwise) would you advise the project to use, and why?
- b. What other measures would you recommend to improve communications within the project?

12.8 Further Reading

12.8.1 Workshops

1. Gottesdiener, E. (2002) *Requirements by Collaboration: Workshops for Defining Needs*, Boston: Addison-Wesley
Gottesdiener's book is a rich source of proven techniques for workshops. It is accompanied by a website which provides free assets for facilitating workshops: <http://ebgconsulting.com/facassets.php>
2. Beyer, H. and Holtzblatt, K. (1998) *Contextual Design, Defining Customer-Centered Systems*, London: Morgan Kaufmann
Beyer and Holtzblatt's book offers many practical insights.

12.8.2 Working in Groups

3. Cockburn, A. (2006) *Agile Software Development*, 2nd Edition, Boston: Addison-Wesley.
Cockburn's is a useful book, full of practical suggestions. It is worth looking at for ideas on group work even if your project is neither 'agile' nor for software.

4. DeMarco, T., Hruschka, P., Lister, T., McMenamin, S., Robertson, J. and Robertson, S. (2008) *Adrenaline Junkies and Template Zombies: Understanding Patterns of Project Behavior*, New York: Dorset House.

Adrenaline Junkies is for background reading, but you may find it illuminating when trying to understand what games people are playing on projects.

5. Boehm, B. et al, *WinWin Spiral Model and Groupware Support System*, <http://sunset.usc.edu/research/WINWIN/index.html>

Tools change too rapidly to track here, but Barry Boehm's work on the WinWin methodology, which was supported by software tools, was groundbreaking, and the techniques should remain relevant.

CHAPTER THIRTEEN

Requirements from Things

Prototypes can be more articulate than people.

Michael Schrage

Requirement Elements	Priorities	Measurements	Definitions	Rationale and Assumptions	Qualities and Constraints	Scenarios	Context, Interfaces, Scope	Goals	Stakeholders
Discovery Contexts									
Introduction									
From Individuals									
From Groups									
From Things									
Trade-Offs									
Putting it all Together									

Answering the questions:

- Given that something exists already, can you effectively rediscover some of the requirements for it, to help develop it further?
- And, given that products are often similar, can you efficiently reuse some of the requirements they have in common?
 - so you have a concrete starting point to work from
 - so you don't waste time 'reinventing the wheel'.

13.1 Summary

Firstly, let us say it at once: requirements are human needs, and come from people. You can't guarantee to get good requirements from things. Things can't check that requirements are correct.

There are exceptions to this rule. For instance, subsystem requirements can, in large part, be deduced from system requirements and design – if those are correct, complete, consistent and up-to-date in every detail. Discovery mainly applies to business requirements.

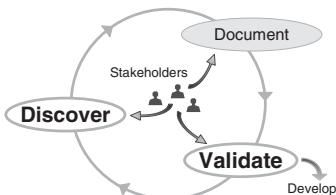
Things – artefacts – can help to suggest requirements in three main ways:

- Requirements prototyping** can stimulate people to agree requirements that the prototypes fulfil. They help people to visualise the consequences of proposed requirements. They can suggest requirements that are missing.
- Reverse engineering** from existing products can recover requirements that were known to other people.
- Requirements reuse** may be able to take existing requirements from suitable sources and apply them to a new product.

You will in every case need to validate the candidate requirements that you discover in these ways with people. Let us look at these in turn.

13.2 Requirements Prototyping

Answering the question: 'So if you don't know what you want, how about this?'



Making prototypes and showing them to stakeholders, in interviews (Chapter 11) or workshops (Chapter 12), is an effective way of discovering and validating requirements, but it must be handled carefully.

13.2.1 Purpose

Prototypes are made for many purposes in systems development, including assessing technology and architecture. Confusingly, the term 'prototype' is also used to mean 'early working version of a product' in evolutionary development.

But requirements prototypes have only one purpose: to improve requirements. Requirements prototypes are created to assist in discovery and validation. They should be designed to be **thrown away** as soon as they have done their job.

It doesn't matter at all what requirements prototypes are made of, whether they do anything or even whether they are 'right', as long as they help you towards better requirements. Making prototypes 'better quality' is usually wrong – it's a waste of money, and can be seriously misleading.

- A good requirements prototype stimulates people to state their requirements better.
- A good prototype may be quite open-ended; you don't know what you need to discover, so being suggestively incomplete may be powerful.
- A bad prototype – such as an excessively complete and beautiful one – could convince stakeholders that the project has already decided on its requirements, and is merely showing off its approach for approval by rubber stamp.

13.2.2 Techniques

Drama is difficult on the page: it's rather like looking at the blueprint for a building rather than the building itself.

David Edgar, playwright

Requirements, like drama, are difficult on the page. Your task is to bring them to life.

Kent Beck, a programmer skilled in prototyping (and a pioneer of agile software development methods), is fond of saying: 'Users don't know what they want until you show it to them'. A prototype is exactly something to show to your stakeholders, to help find out what they want.

Further, requirements often make sense only in a context: a product, a system, a business environment (see Chapter 4). A prototype, and perhaps a prototyping workshop or demonstration, forms a solid, tangible context. A good prototype triggers a rush of requirements as people start to visualise what they could possibly have in that context.

An inquiry is a collaborative venture in which a team of people work out a shared point of view. Inquiry typically proceeds in a cycle because ideas get

Prototype Screens Reveal Data Deficits

by Dr Frank Houdek, Daimler

Prototypes can bring abstract requirements close to the intended working environment. For one project, we developed a new product documentation tool for automotive electronic control units (ECUs).

In this context, hundreds of data items are relevant for the various data customers (e.g. cabling needs pinning information, power supply needs information on power consumption, electromagnetic compatibility (EMC) needs information on electromagnetic emissions, ...). The supposedly complete list of relevant data items (written down in a spreadsheet) had been reviewed several times by the data customers.

Then, we built a user interface prototype clustering the data items into screens. Reviewing these screens not only uncovered problems in the screen design (as intended) but uncovered deficits in the previously agreed data items as well, just because now the data items were seen in a much more concrete context.

Reproduced by permission of Dr Frank Houdek.

more definite as you go along, so you need to revisit vague early assumptions and update or correct them. Every development project is an inquiry to some extent. In requirements prototyping, inquiry is the dominant activity.

To use prototypes to discover requirements, you simply run a miniature inquiry cycle (the quicker the better), as in Figure 13.1. Each time round the cycle, you find out a little more about what is wanted. When the stakeholders are happy and you have found out what you need, it is time to stop.

The prototyping cycle has three phases:

1. Listen carefully.
2. Make a prototype.
3. Demonstrate the prototype.

This cycle is a concrete expression of the discover-document-validate inquiry cycle (see Chapter 1) used throughout this book.

Let us step through these three phases of the cycle in turn.

Listen Carefully

Your purpose is to find out what the stakeholders want, not to show how clever you are at prototyping.

People state requirements very quickly on seeing a prototype. They will typically say most of what they have to say in the first two minutes.

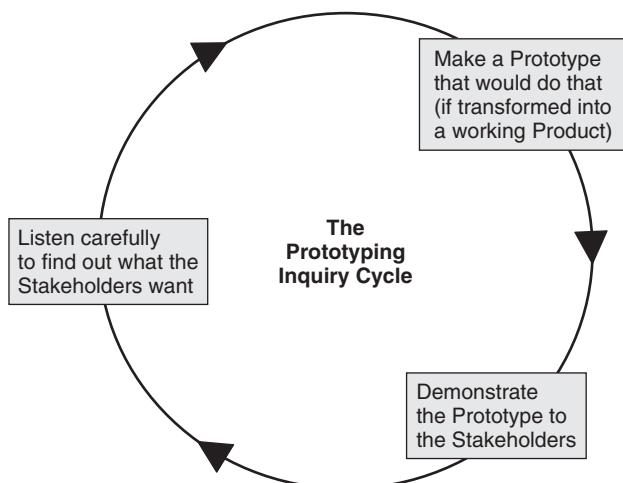


Figure 13.1: The prototyping inquiry cycle.

People can react instinctively to a prototype, e.g. if it has a style or appearance they do not like. Do not be offended! You are finding out something quickly and cheaply that would have cost you a lot of time and trouble if you'd built the final product like that, and had to change it. People react strongly because:

- they suddenly understand what it was that your team has understood them to have said;
- they also suddenly understand a little bit more about what they want!

Your task is to:

- listen to what your stakeholders say; and
- watch their faces and body language to understand what they feel.

It is quite difficult to demonstrate a complicated prototype and to watch and listen at the same time. So:

- Either make your prototypes very simple or have one person do the demonstration and other people to watch, listen and record the stakeholders' reactions.
- You should consider how to record what happens (see the section on observation in Chapter 11 for a detailed list of options). An audio or video recording could be helpful, as you only get one chance to get people's first reactions to a prototype.
- If you are new to prototyping, you might want to practice demonstrating your prototype and recording people's reactions on some friendly colleagues first: prototyping your prototyping process.

A good requirements process is full of challenges and surprise discoveries, leading to a smooth and problem-free development and product launch. So, it is a success if you discover lots of issues with your prototypes.

Make a Prototype

*Making is a form of learning that combines the tacit with the evident
in a way that no other thing can.*

Antony Gormley, sculptor

Making a prototype can help you to discover tacit requirements easily and quickly. Grappling with actually making something brings you face-to-face with issues that words don't reach.

A requirements prototype can be made of almost anything [2]:

- Cheaper and quicker is usually better than slower and more costly.
- Several simple prototypes are usually better than one complicated model.

All a requirements prototype has to do is give people the idea, so you can ask: 'Is this right?'. Polish and perfection tend to get in the way. Sketched, human and playful are desirable qualities in an early prototype.

Some useful ways of building requirements prototypes are shown in Table 13.1, along with advantages and disadvantages of the different techniques.

Whatever you choose to do, remember that your purpose is discovery.

Demonstrate the Prototype

The first law of demonstrations is that anything that can go wrong, will. There are no prizes for creating complicated models that nobody can get to work. So:

- keep the demonstration simple;
- make sure it 'works' as you intend;
- if necessary, write a script for your demonstration.

This does not mean developing lots of software for error checking – that is usually a bad idea. It just means that you know that you can show people what you are thinking by stepping through a simple story with the prototype.

For example, with a paper prototype, the demonstration may be that you just say a few words about 'screen 1' (the first sheet of flipchart paper), pretend to press a button, and then flip over to 'screen 2' which correctly shows what should appear when that button is pressed.

Always run through your demonstration in advance (even if you are not going to rehearse the rest of the process). Just as a prototype is cheaper than a

Table 13.1: Some techniques for building requirements prototypes.

Construction technique	Advantages	Disadvantages
Coloured marker pen drawings on flipchart paper	Quick, informal (Possible at any time in an interview or workshop, if you need to explain an idea)	May need more accurate mock-ups later
'Screen' images drawn as diagrams on presentation slides (e.g. PowerPoint)	Gives an idea of how the system might feel	May focus attention on user interface design
A hypertext of screen images, linked so that 'button clicks' navigate to the correct screens	Navigable, gives an idea of how the system could fit together	May focus attention on navigation design
Circles and rectangles of coloured felt placed on a blanket draped over a flipchart, to form an easily-reconfigured mock-up screen	Invites people to join in redesigning the prototype, ideal for user interface design	Again, need to remember other types of requirement
A cardboard mock-up of a console or handheld device, decorated with photographs of screens, dials, switches etc	Quick, informal; lets people check they can reach the switches, etc	May focus attention on human factors
A screens-only prototype developed with a user interface design tool, providing some navigation but no software in the button callback functions	Gives a realistic impression of how the system might look	Could mislead people about project progress
A scene acted by people from the team playing the roles of users, the product, etc. Can be supported by a paper mock-up or other props	Teambuilding; gets everyone involved in thinking about scenarios	One-off; not available for reference later
A video of an acted scene or sequence of scenes, set up to look as if the product was already working	Helps people to imagine how the product might work	Could give impression that background details are required rather than incidental
A software implementation or simulation of one function of the product, possibly using a prototyping tool	Lets people actually try out the product	Could divert resources from discovering other requirements
An old version of the product, or a competitor's product, or an analogous product from a different field (perhaps combined with an acted scene, etc)	Gives an idea of what could be possible; may stimulate creative thinking	Could lead in an unproductive direction

finished product, so a dry run is less embarrassing than a spoilt demonstration. The same principles apply as for a workshop (see Chapter 12), so make sure that everything you'll be using in the room is as you want it and works properly.

Demonstrating a 'Human CPU' Prototype

by Dr Frank Houdek, Daimler

We had a good experience with paper mock-ups presented in a three-person environment:

- The first person is a stakeholder who sits in front of the system and uses it to carry out some tasks. The tasks were explained to her previously, e.g. entering a new invoice and checking product stock.
- The second person is the 'human CPU' who reacts to the stakeholder's actions. If the stakeholder 'presses a button' (by pointing to the hand-drawn button on the current screen), the human CPU presents the appropriate follow-up screen.
- The third person is the observer recording the stakeholder's comments.

This simple setup revealed many requirements.

Reproduced by permission of Dr Frank Houdek.

Each run of a prototype, with software or without (e.g. animated by people in a workshop), effectively simulates or explores one scenario (see Chapter 5), i.e. one story in which somebody uses a product for a particular purpose.

Have People 'Use' the Prototype

For a user interface prototype, take a scenario and ask a user to walk you through how they'd imagine they would complete that scenario with these screens. Even with paper screens, they can point and talk about the scenario. A variant of the 'acting scenes' approach is discussed in Chapter 5.

Remote Demonstrations

Demonstrations are traditionally done in person, face-to-face. As projects face increasing time pressure and work across more sites (even with teams split across time zones and oceans), people wonder if they can create requirements without meeting each other at all.

But remoteness immediately means formality. Even with the best technology, such as video links and meeting software, the 'bandwidth' for communication is much less than a face-to-face meeting. For example, you

don't get the atmosphere of the meeting, and you may miss body language (especially of nonspeakers). You may get improved discipline, as only one person can talk at a time, but you may miss out on other people's requirements.

In general, therefore, demonstrations should be held as physical workshops. If people have to travel very far, you will try to pack as much into each workshop as possible, perhaps with several prototypes, briefings and other activities. That unfortunately means you will have fewer meetings, which makes prototyping more difficult and less effective. You might do better to hold some demonstrations remotely (using video conferencing); that way, you get more prototypes and, at least sometimes, the benefits of meeting the whole team face-to-face.

Arguments for and against holding demonstrations face-to-face and remotely are listed in Table 13.2. See also the discussion of groupware in Chapter 12.

Demonstrating Software

Do not immediately jump into creating software prototypes for discovering requirements. Software is very useful for many types of prototyping, such as discovering whether a mechanism is feasible or predicting performance, but it is not necessarily the best tool for making requirements prototypes. If you can

Table 13.2: Live versus remote demonstrations.

Demonstration method	Advantages	Disadvantages
Face-to-face workshop	Immediacy, wide 'bandwidth' of communication with stakeholders; many-to-many-person interactions; body language; ability to iterate rapidly with facilitation and reach agreement with whole team	Cost of travel and time, especially if you have many stakeholders and your project teams are on widely separated sites
Remote, using video, meeting software, etc	Lower cost, less travel; can demonstrate prototypes more often, if remote meetings are easier to schedule	'Bandwidth' of verbal and nonverbal communication may be inadequate; may be hard to get a proper impression of the prototype; important issues may not surface; difficulty in scheduling meetings across time zones may make meetings too infrequent for efficient project working

find a simpler way to show what you mean, do that instead. Table 13.1 lists numerous simple alternatives, all of which have proven effective on projects.

If you have no alternative:

- ✓ Keep the software simple: only prototype the features you need to demonstrate. Often, there is no value in actually making the software do anything ‘behind the scenes’ such as processing data or writing files; a ‘screens-only’ approach may be best.
- ✗ Do not waste time on error-handling.
- ✗ Do not get dragged into user interface design: you are only showing a rough user interface because you are interested in discovering the product’s functions and interactions with the user. You may need to say this explicitly at the start of your demonstration.
- ✗ Make it clear that this is just a simple prototype with very limited functionality and almost nothing behind the scenes (no communications protocols, no database, no error trapping, etc). You should say this explicitly, before the demonstration begins. You are trying to show that something could possibly work, not that you have finished engineering a solution.

Worked Example: Demonstrating with a ‘Screens-Only Mockup’

Here is a deliberately rough, screens-only prototype (Figure 13.2) of a software product to identify which candidates for university entry should have their qualifications accepted:

- it is just a picture of how one function of the product might work;
- the user interface design is not being enforced;
- no effort has been spent on aspects such as navigation, help or branding.

The prototype allows you to check with stakeholders that you have understood the domain and its rules correctly, even if no software is actually present.

However, even a fairly rough prototype like this has its dangers:

- stakeholders can easily take the sketchiest design as more or less definitive;
- stakeholders may fall into the ‘rat hole’ of discussing minuscule details of user interface design such as fonts, alignment, colours, box widths, button labels and so forth.

To reduce the risk that stakeholders take your prototype as the chosen design, you could:

- make a hand-drawn paper prototype (Figure 13.3);
- scan hand-drawn screens, rather than letting a tool make them too tidy;
- use a ‘handwriting’ font and deliberately rough boxes (e.g. using a ‘free-hand’ or ‘scribble’ line in your drawing tool, rather than neat rectangles).

Prototype Enrolment Qualification Checker X

Candidate:	Andrea	Course:	P-053 Physical Sciences	▼
Qualifications:				
A-Levels	Subject	Grade / Mark		
	Physics	▼	A	▼
	Chemistry	▼	B	▼
		▼		▼
		▼		▼
Other	▼		▼	

Check

Figure 13.2: Simple mock-ups can help people to decide what a product should do.

Even with this degree of roughness, stakeholders may still be tempted to suggest trivial user interface design requirements, e.g. that the fields should be lined up. At some stage, those design details will be relevant, but if capturing them is not your purpose, then make it clear at the start that you are primarily interested in discovering, say, what functions and data are needed.

A sketched screen mock-up like this example, combined with a table of rules, can be ‘executed’ by stepping through a scenario manually with stakeholders: ‘OK, let’s see if Andrea would qualify to study physical sciences.

1. You put Andrea’s mathematics A-level grade in here ...
2. Then you push the ‘Check’ button.
3. The system looks up the table ... here it is ...
4. ... and tells you that Andrea can be enrolled.’

A prototype shows stakeholders how you are progressing, and that you are taking their suggestions seriously, as well as enabling you to improve the requirements.

Prototypes can also stimulate stakeholders to comment on the value of tools to support particular business functions, such as enrolling students.

Quick and Dirty

Experts, gurus and ordinary folk have emphasised time and again that exploring and making mistakes, quickly and cheaply, is the right way to improve.

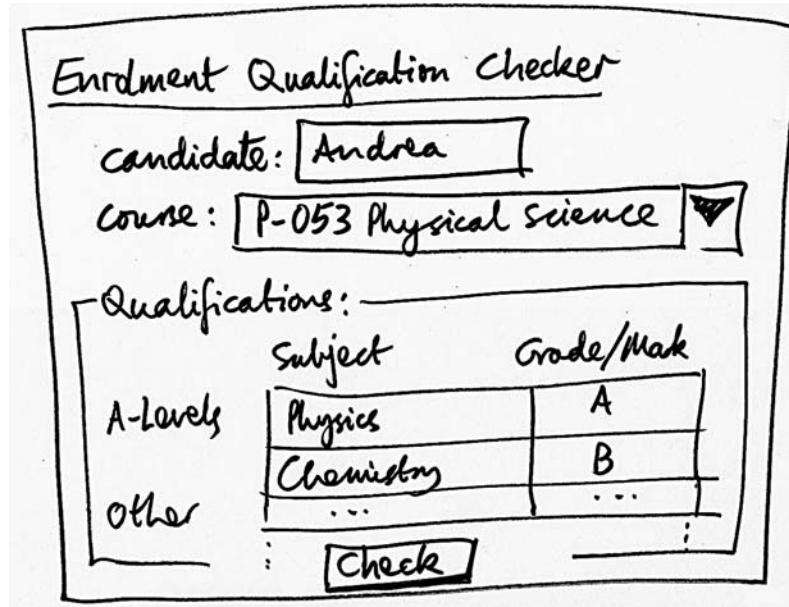


Figure 13.3: An intentionally rough, hand-drawn paper prototype screen.

But somehow, it seems a hard lesson to learn. So, at the risk of repetition, here are the key points about iterating your requirements (again):

*If a thing's worth doing,
it's worth doing badly*

Richard Stevens

Serious Play

Michael Schrage

*Err and err and err again
but less and less and less*

Piet Hein

Your goal is not to design systems for specific functions (such as checking qualifications) and still less to create software, but to get the requirements right. A good prototype may be of low quality (with respect to the designed product).

In his book *Serious Play*, Michael Schrage (1999) [1] argues that the ability to make more prototypes more quickly – making more mistakes, if you like – is the main force that has propelled modern business.

Valuable information can be gained from prototypes made of almost anything. Traditionally, cars were prototyped on paper (for initial concepts), in clay (for detailed styling) and in wood (for aerodynamics).

It is software, of course, that has enabled people to make powerful and complex prototypes of cars and jet engines and high-tech buildings and everything else.

Software makes prototyping quick and cheap enough to serve as a practical alternative to trying to get everything right first time: the old 'big bang' (often wrongly called the 'waterfall') development life cycle.

In requirements work, early in the life cycle, prototypes can be very simple and rough indeed.

Tips for Requirements Prototyping

DO

- Make plenty of simple, quick, cheap prototypes.
- Tell people what you are about to show them: it's only for one aspect of a product, or it's a revised prototype incorporating their earlier comments, etc.
- If you're showing a screen mockup, make it clear that actual screen design is for later, and this is just to discover required functionality.
- Show prototypes to different stakeholders.
- Make sure you capture (e.g. with video) what people say about a prototype in the first minute or two – the moment won't happen again.
- Watch people's faces as you show them each feature of your prototypes.

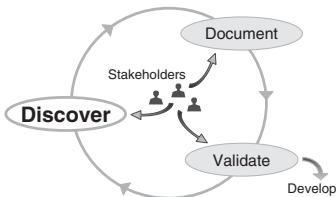
DON'T

- Don't be afraid to sketch out ideas for what a stakeholder's stated requirements might mean in practice:
‘Is this the sort of thing you'd like?’
- Don't waste time using elaborate prototyping tools if all you need is a quick sketch.
- Don't make a prototype into a beautiful, endlessly smartened-up project deliverable; that's a waste of time, and a distraction.
- Don't bow to pressure to turn a throwaway prototype into a product ‘to save time’. Their purpose, structure and quality are entirely different (you might choose a mockup technique for this reason).

13.3 Reverse Engineering

Answering the questions:

- Do many of the requirements exist already, embodied in legacy or competitors' systems?
- Are there useful ideas in analogous systems of other kinds?



This book emphasises the creative aspect of requirements work but, as in the world of art, creation is rarely done from scratch. Engineering proceeds by extending, mutating, fusing and transforming the best existing ideas.

Requirements can be found by examining existing systems, whether these are legacy software or competitors' products, or other domains altogether.

'Archaeology', the process of digging through old specifications and user manuals, can also suggest reusable requirements. All such ideas must then be validated with stakeholders.

13.3.1 From an Existing Product

One of the most common scenarios in product development is the enhancement of a product that has enjoyed some success. This may be because:

- the product has suggested new possibilities to its users;
- the market has moved on, and people expect new capabilities;
- problems have been found, and must be fixed.

'Brown Field' versus 'Green Field' Development

Whatever the reason, this means your project will be starting from a 'brown field site' with something already built on it, rather than some idealised fresh 'green field' in which you can freely create whatever takes your fancy. In fact, the brown field is much more likely, simply because most products are created only once but modified many times.

You might imagine that you wouldn't need to do any discovering on your own product. But teams change; staff leave; documentation gets out of date, or doesn't get written; and systems often 'just growed', like Topsy.¹ So discovery is definitely something to plan for.

¹In Harriet Beecher Stowe's novel *Uncle Tom's Cabin*, the character of Topsy is asked where she came from. She replies 'I s'pect I just growed. Don't think nobody never made me.' Plenty of systems have no discernible plan or specifications, either.

Requirements on a 'Brown Field Site'

The key to success when you're working on a 'brown field site' is to start from where you are, not where you'd hope to be. If the ground of a building site is uneven and unsafe, the demolition men level it, or excavate until it's safe to start. In other words, changing an existing system involves extra work.

What you need to start your requirements when there are numerous constraints already in place is to find out:

- **what is there already:**

- existing interfaces;
- existing components;
- existing manuals and training materials;
- existing simulators and test equipment;

- **what will certainly have to be changed:**

- interfaces you know are obsolete;
- functions you know are missing;

- **what has to be assumed:**

- e.g., that certain components can continue to be used unchanged;

- **what risks you are taking:**

- e.g. that existing components will have to be modified (i.e. they need to be redeveloped at extra cost),
- e.g. that important but old types of component, such as processors or tape drives used in the old system, will continue to be available from manufacturers.

Very likely, nobody exactly knows what the old system does in every situation, especially when it's trying to handle combinations of errors. Experimenting may be exactly the wrong approach. You could be better off thinking out what ought to happen, rather than wasting time finding out what a tangle the system used to get itself into.

On the plus side, the existence of a system you can look at may make it much quicker to get up to speed with what is needed, and to ask well focused questions.

All of these things make a brown-field project feel quite different from classical green-field requirements discovery. Indeed, you may never need to interview people to find out what they want: that may not be the problem at all.

Awkward choices are likely:

- Will you decide to leave some components alone, even though they are not perfect, because you can't do everything at once?

- Will you replace some parts altogether, even though they do work?
- Will you modify some parts (instead of replacing them) because they are almost good enough, and accept the risk that this could turn out to be harder than starting again?

There are several ways you can discover requirements when a product already exists, including:

- from an earlier version of the product;
- from simulators;
- from problem reports;
- from product documentation.

Let us look at each of these in turn.

From an Earlier Version of the Product

With a working but poorly-documented product that you are supposed to upgrade, you can systematically explore and document its functions and performance, e.g. by trying out all the menu options on a piece of software. This may be a substantial task. Working out an implied business process, given a software product that perhaps only partially supports it, may be difficult. If you find bugs, you have to decide whether to fix them and, if so, what the correct behaviour should have been.

If there is little or no documentation, it may require substantial effort to discover the effect and purpose of the more complex options provided. For example, you may need to develop test data files for a software product to process; you may need to locate old test harness and simulators – or develop new ones – to explore interactions across interfaces.

The circumstances in which you may need to explore a product's behaviour are diverse, so advice must be rather general. Since you could waste a lot of time finding out things that are no longer useful, you must have a plan and a purpose, e.g.:

- to (re)discover which business processes are supported; or
- to identify good ideas for ways to support specific tasks, etc.

From Simulators

Large commercial, government, military and space systems often have simulators that accurately reproduce the behaviour of in-service products. These may have been intended for training or mission planning, to provide a realistic environment safely separated from the operational system.

For example, in a chain store's retail system, shop assistants are trained on tills exactly like the real ones, except that payments do not really get taken from the 'customer's' credit or debit card, and the stock changes do not cause actual goods to be ordered from suppliers.

If you are lucky enough to have such a simulator, you can use it to explore product behaviour just as with the product itself, without the risk of causing any harm.

From Problem Reports

Existing problem reports and similar documents, such as complaints and suggestions from users, can directly suggest possible new requirements. You need a good understanding of the product to be able to evaluate problem reports.

You will certainly need to prioritise any list of proposed requirements that you come up with (see Chapter 10).

Where you have a community of users who are continuing to use the most recent version of a product while you are developing the next version, recent problem reports are a major source of requirements, both for upgrading the product and for correcting errors.

This is such a standard activity that software and systems engineers tend to think of it as a separate process, change management (Figure 13.4), rather than part of the work of discovering requirements.

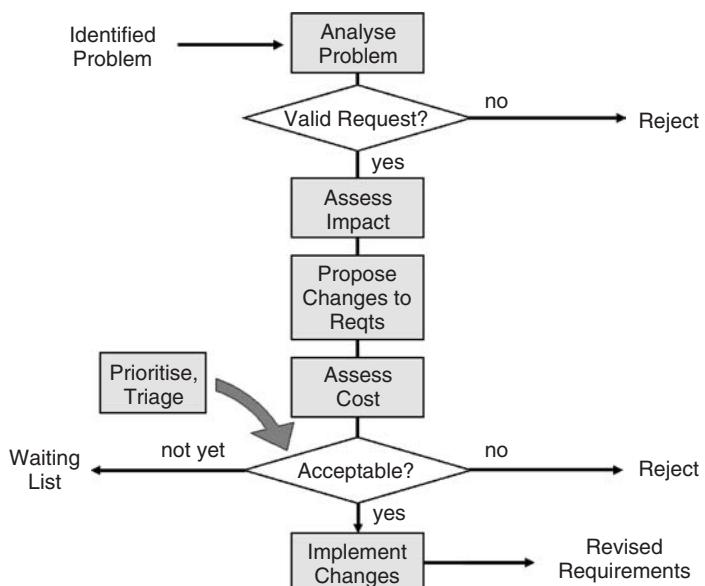


Figure 13.4: The change management process: turning problems into requirements.

The amount of initial problem analysis you need to do will vary with the quality and detail of the inputs you receive. Problems are typically reported via several channels:

- **Software problem reports** filled in by programmers or test engineers are likely to be accurate and detailed. They most likely indicate things that urgently need fixing (to meet existing requirements) and they may often also suggest missed or new requirements (which need to be prioritised).
- **Suggestion forms** will be of variable quality. Some will be directly useful.
- **Review comments** (when colleagues on a project examine documents prior to their release) will vary from apparently trivial comments on document layout and punctuation, right up to crucial discoveries of missed requirements. Brief comments from good reviewers – both knowledgeable in the domain and skilful at reviewing – can be incredibly valuable.
- **Customer requests** (to your helpdesk or technical support) will certainly reflect perceived problems and perceived needs. These may not necessarily concern your actual product; for instance, the issue could be training, documentation or other software. However, when a customer perceives a need, you have a real opportunity to offer a solution. That very possibly includes improvements to your product. You need a systematic mechanism to identify ‘real’ issues behind requests.

From Product Documentation (Archaeology)

Archaeology is the somewhat ironic name given to the work of digging through stratified deposits of old product documentation (see Figure 13.5). As the name suggests, the task can be hard work but, in particular:

- old documentation may explain the rationale for features, perhaps in the form of information meant for suppliers or maintainers, that would be hard or impossible to guess;
- lists of exceptions and error messages are valuable, as it is hard to create the conditions to trigger all of them if you don’t know what they are.

There are several dangers in seeking valid requirements from old documents:

- The context assumed in old specifications will probably be out of date. As other components have been added or modified, the way that products behave will often be different.
- Documents may contain mistakes that would have been evident to product designers and users, but may now be obscure.
- Software, especially, is often modified by maintenance programmers, who may not update the specifications to match.

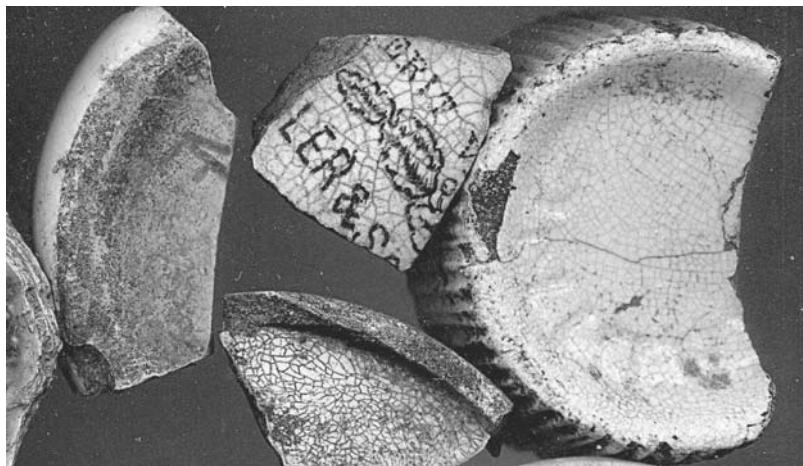


Figure 13.5: Archaeology: piecing together requirements from old documentation.

- As documentation is usually updated sporadically, there may be many versions of some documents – just like layers of broken pottery in a real archaeological kitchen midden. It may be difficult to determine exactly which version of a requirement applies to which version(s) of the system.²
- Documents with a sales function (even if technical) may not be completely open about weaknesses in a product. For example, user manuals often sidestep any shortcomings in error handling.

But again on the plus side, at least you have something to give you an idea of the context before you start talking to users.

Therefore, you should not expect too much from ‘archaeological’ work. If possible, check any findings with people who remember the product, or try them out on the product itself. Finally, the fact that something used to be a requirement does not prove that you need it today: evaluate and prioritise.

From Competitors’ Products

Comparing your ideas with your competitors’ is obviously a sensible thing to do in a free market; indeed, it’s a basic survival strategy. A competitor is a classic example of a stakeholder who doesn’t want to share his requirements with you – a military enemy is another. Your interests are best served by finding efficient and economical ways of discovering what your competitors are up to.

²If the previous project used a Wiki or other searchable tool to track changes (see Chapter 12), then archaeology becomes much easier. ‘To benefit the next project’ is a reason to use tools but, from the project manager’s point of view, it is always secondary to getting the immediate job done.

We can't encourage you to go in for industrial espionage, but we quite understand the temptation. Short of that, you can look at what's on the market, buy a sample, read its manual and play with the product. You shouldn't need to do anything nefarious like disassembling the compiled software to discover candidate requirements; features of interest to a user should, after all, be discernible from the user interface.

You can also read the technical press for clues about what plans are being hatched, and keep your ears open at trade fairs and conferences.

The key thing is perhaps to look out for innovative ideas.

From Analogous Products or Concepts

You can use an analogous product from another field to stimulate people into contributing ideas by reacting to what they see, i.e. as a kind of prototype. You can equally well use a metaphor (see box, 'Example Analogy: A Lending Library') to get people to explore ideas for a product.

Deliberate use of analogy requires care and skill, as people may not see the relevance of what you are doing. Many people take examples literally. They have difficulty making the imaginative leap from one domain to another. It may help if you introduce the analogy session by saying something like:

'I am now going to give you an example of a product from a different industry. I would like you to see if you can find any analogies between what we are doing here, and what the designers of this product were doing over there.'

That way, people's natural reluctance to look at something 'irrelevant' can be overcome, and they will start to come up with valuable requirements.

Example Analogy: A Lending Library

For example, you might ask people working on a computer-aided design (CAD) product to explore the metaphor of a public library. You could ask:

'How are new users registered by the library? Do they need to have their parents' approval? Who are the "parents" in the case of CAD users?'

'How does the library track who has borrowed a book? Is checking out a CAD diagram like borrowing a book? What tracking capabilities do we need for CAD diagrams?'

and so on.

Analogous products and concepts can, in this way, be used to discover both functional features (see Chapter 5) and qualities (see Chapter 6).

The use of analogy is a creative technique that, in the right workshop conditions, can help people to reflect on what they want and what features could possibly be helpful in the product.

Tips for Reverse Engineering, Archaeology, and Analogy

- Try a range of sources of possible requirements.
- Ask people who are familiar with the product to suggest sources.
- Play your findings back to knowledgeable people.
- If your favourite technique doesn't work, try something else.
- Keep a list of rejected requirements, and a waiting list.

13.4 Requirements Reuse

Answering the questions:

- Can we save time and money by reusing some of this stuff?
- Have we got anything that does XYZ?
- What does this product do that we might reuse?

Requirements reuse isn't easy. In a nutshell, it does not make much sense to try to reuse individual requirements without context. Logical groups of requirements, such as use cases, are more promising.

Three main types of requirements reuse in practice are:

1. naïve reuse;
2. standardisation;
3. product lines.

Let us consider each of these in turn.

13.4.1 Type 1: Naïve Reuse

People look out a 'similar' project and try to cut and paste whatever they feel they can reuse. Unfortunately, this breaks the original structure and traceability (if any). The work of reuse may, in the worst case, be as heavy as starting again.

But, depending on the type of product, naïve reuse is often quite a good strategy. If the new product is very similar to the old one, copying and modifying may be the quickest and cheapest thing to do.

13.4.2 Type 2: Standardisation

People create, and other people refer to, books of requirements that apply to all projects in a domain. These include:

- regulations;
- standards;
- industry best practice.

You can think of these as the requirements that people know must be applied, regardless of the functions in an individual project within the domain. The problem is that applying every possible standard to a project is very costly.

Examples of how to call up standards and regulations in your requirements are given in Chapter 6.

13.4.3 Type 3: Product Lines

People mark up a ‘reusable’ template requirements document with sections that apply to projects with different characteristics. When you start a new project, you go through a series of questions to characterise your project (e.g. it involves safety, there are electrical devices, it will work outdoors). Then these characteristics are used to filter your template, and you have the kernel of your specification, ready-tailored to your project. The problem is that making the reusable template (probably from the outputs of several earlier projects) is a substantial project in itself, and it can still go out of date.

Both standardisation and product line engineering help mainly to reuse ‘nonfunctional’ requirements. However, in a well-defined domain, basic functions, too, may be reusable. There is certainly scope here for well-documented patterns.

13.4.4 Tool Support for Reuse

Reuse can clearly be supported by suitable tools, depending on what you want to achieve. Type (1), naïve reuse, needs nothing more than a software cut-and-paste clipboard. Type (2), standardisation, can be well supported by requirements management tools, by an ordinary database or by a document management system.

Type (3), product lines, demands more specific support (see box, 'Case History: Building a Special-to-Purpose Reuse Tool').

Case History: Building a Special-to-Purpose Reuse Tool

The task was to extend a requirement management database with a customised tool to provide for product line reuse in a manufacturing company.

The company prepared a database containing all the variations of standard features and components that they expected their product line to contain. For example, there were electric motors operating at different voltages and of different types. The database was provided with fields defining the ways in which these features and components differed: for example, a motor could be linear, inductive, etc.

The company then prepared a table containing a set of questions to take an engineer through the task of specifying a new product. The idea was to have the tool ask question 1. On being given an answer, the tool would record this, and decide whether to ask question 2 or question 3, and so on. For example, if the component chosen was a motor, then the next question would be about the type of motor.

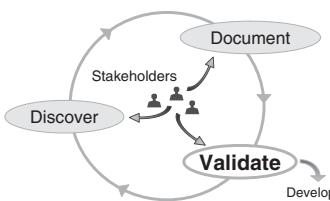
The tool, therefore, had to read the relevant question and the allowed answers, and display them in a suitable window. When the user picked an answer, the next relevant question had to be identified, the window had to be replaced with a fresh one for the next question, and the data recorded. The questions formed a branching tree of possibilities, and the set of answers given by the user defined a desired new product. At the end of the questionnaire, the tool had to select all the relevant parts of the database, copy them to form a new document, and populate its fields with the values supplied by the user.

The result was the semi-automatic production of a document, as from a template, ready-populated with all the usual 'boilerplate' text, but with only the exact sections needed for the new product: very convenient.

The main limitation of a tool like that is that, when the world moves on, the database has to be updated to keep abreast of changes in technology. Old features become obsolete and new ones appear; so (be warned) the approach is more useful for qualities and constraints than for functions.

The tool is worth maintaining if the saving of effort (including the savings from prevented mistakes) when setting up the specifications for each new product is less than the cost of keeping the tool up to date.

13.5 Validating Requirements from Things



The main danger when discovering requirements from things is that they may not be what stakeholders want. So, your main task in validating such requirements is to get stakeholders to agree or reject them. There is not much point in detailed modelling without stakeholder buy-in. This might involve interviews or workshops.

13.6 The Bare Minimum from Things

Make sure you know enough of old systems to avoid reinventing the wheel.

13.7 Exercises

Your company is planning to make a multifunctional device to help people on active outdoor leisure pursuits like mountain walking, camping, fishing and canoeing.

1. Gather information about existing competing products (e.g. using the Internet).
2. Make a paper or cardboard prototype of your product.
3. Show your paper prototype to a colleague. Record their comments.
4. Use the comments on your first attempt to make a better prototype.

13.8 Further Reading

13.8.1 Prototyping

1. Schrage, M. (1999) *Serious Play: How the World's Best Companies Simulate to Innovate*, Boston: Harvard Business School Press.
Michael Schrage's book sings the praises of playful and effective prototyping. It is written for business people rather than developers.

2. Snyder, C. (2003) *Paper Prototyping: The Fast and Easy Way to Design and Refine User Interfaces*, San Francisco: Morgan Kaufmann.

Many books have been written about prototyping. One that may be helpful in persuading your team that prototyping can help them is Carolyn Snyder's sensible, practical and well-illustrated book. It is strongly focused on user interfaces and usability testing, which from a requirements discovery point of view is just one approach, but certainly a powerful and relevant one.

CHAPTER FOURTEEN

Trade-offs

Thoughts are but dreams till their effects be tried.

Shakespeare, *Lucrece*, 353.

Requirement Elements	Priorities	Measurements	Definitions	Rationale and Assumptions	Qualities and Constraints	Scenarios	Context, Interfaces, Scope	Goals	Stakeholders	From Things	From Groups	From Individuals	Introduction	Discovery Contexts
Trade-Offs														
Putting it all Together														

Answering the questions:

- Which design option best meets the project's goals?
- Which requirements can you implement, given that design approach?
 - so you get a product that meets the requirements as well as possible
 - so you agree on a realistic, affordable solution.

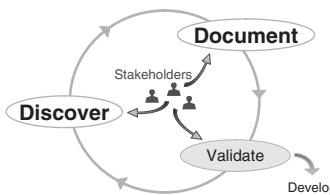
14.1 Summary

Trade-offs identify possible design options, and evaluate which option best (but still imperfectly) meets the (input-prioritised) requirements. The process is creative; discovering the best match of design and requirements cannot be fully automated, though several techniques can clarify the decisions to be made.

The result is a list of requirements that are known to be implementable and, at the same time, a design approach that is known to be able to implement those requirements.

Once the design is chosen, the project can determine its output priorities for implementation, and can analyse the requirements to be placed on developers or contractors in detail. A similar process can be used to select a supplier.

14.2 Optioneering: The Engineering of Trade-offs



14.2.1 The Requirements-First Life-Cycle Myth

Requirements books and courses usually proceed as if you first do the requirements, then the implementation, then the tests, in a one-pass, fixed, 'waterfall' life cycle.

Remarkably, projects still sometimes try to define requirements with no mention of design, which shows the power of dogma over reason. On very small projects, this approach may possibly succeed (though we suspect only by hidden iteration and rework). On larger projects, it probably always fails, except where the requirements and design are already well known through having been tried on earlier projects. Fortunately, there are experienced requirements people in industry who know better; see Ralph Young's guest

box, 'Iterate Requirements and Design Repeatedly'. Young's books (e.g. [2]) provide much more detailed advice on the subject.

14.2.2 An Optioneering Life Cycle

We need to understand the stakeholders' intent, then produce a solution which addresses that intent to the best of our ability.

Scott Ambler

A more realistic life cycle (which should be tailored to your project's circumstances) begins by discovering stakeholders' requirements. Then a basic design is sketched out, simulated or prototyped, which, of course, means that the product is designed and tested in a small way. Most likely, many problems are discovered. Often, several competing design options emerge. These may be overlapping variations on a theme, or may be sharply distinct from each other.

Depending on who has the knowledge – you or your suppliers – you may think out the design options for yourself, or different suppliers may each propose a design in response to your requirements. Either way, you have to choose which design you want; in the latter case, you are also choosing a supplier. That can be a major decision, with long-term consequences: the supplier will try to use his knowledge of the design to lock you in to using him for all future modifications and developments.

The design options may all meet some of the requirements well but, usually, each option fails to meet certain requirements. This is the world pushing back; and it is the requirements that will have to give. You will probably, as Ralph Young suggests, have to loop between requirements and design several times until you reach a satisfactory compromise.

Trade-offs can be made at any level, from the basic architecture to use for whole systems, down to subtle choices and optimisations at subsystem or component levels. This chapter describes a single optioneering cycle; you may need to apply the process repeatedly during development.

Now you face an engineering challenge that is barely described in the requirements literature: choosing which requirements *not* to satisfy completely. To put it more precisely, you need to choose the design option that best meets the requirements:

- Any option that cannot meet a genuinely essential requirement must be discarded. Logically, if all the options fail to clear this hurdle, the project must be cancelled or at least rethought.
- When there are many requirements and few options to choose from, you may label only a few requirements as essential if you want any of the options to survive.

Guest Box: Iterate Requirements and Design Repeatedly
by Ralph Young, practitioner and author

Any project is well advised to use an iterative approach to evolve the real requirements and to determine the design for the capabilities to be provided. Aspects of this include:

- We know from our experience that the requirements are *never* finished or finalized – indeed, new requirements and changes to requirements are always suggested continuously throughout the development of a capability.
- The iteration should be completed in a finite period of time – either we arrive at something workable or we recognize that we haven't scoped a capability yet.
- 'Better requirements' refers to real requirements that meet criteria for a good requirement.
- Consider the 'designability' of the capability when addressing the requirements. Since we are evaluated on our ability to deliver actual capabilities on time and within budget, the first order determinant of development costs/time will be the product specification. Red-flag requirements found to be troublesome, and identify a team to investigate trade-offs (resulting in the incorporation of new technologies or tools, or just safer ways to amortize risks via process steps, for example, through early prototyping). Do a cost versus benefit trade study from the user perspective. By doing design trades concurrently with requirements development, we can largely avoid the catastrophic effects of having to cull the bad requirements from the real requirements late in the development program.
- Allocate requirements to functional partitions, objects, people, or support elements to support synthesis of solutions. Utilize a system architecture process.
- Use open systems standards. An open architecture implies lower risk that the evolution of the capability will result in major revisions of the product architecture. Another purpose in seeking compliance with open systems standards is to achieve independence from proprietary standards of vendors. The standard interfaces make it possible to buy components from any supplier with standard products, making the capability easier to buy, easier to maintain, and more flexible and scalable.
- Consider utilizing recommended guidelines for architecting. An iterative process is needed because additional information impacts

the known requirements, the architecture, and the design. Examine all assumptions to determine whether they need to be allocated to lower levels of the architecture, and allocate downward if appropriate. Use care when suggesting a solution if you suspect that all of the requirements are not known. Apply risk management techniques.

- Never assume that the original statement of the problem is the best one or even that it is correct.

Reproduced by permission of Ralph Young.

- If the winning option cannot meet some goals in full, then those goals must be modified or discarded, even if their input priority is very high.

You can see that you don't really know, definite requirements until this process is complete. As Scott Ambler says, the thing you are trying to satisfy is the stakeholders' intent – a set of goals – and the best design will only approximately do that. It's like an optimisation problem, but with many dimensions that are hard to compare.

Optioneering is a key activity when contracts are being set between customer and supplier: the customer may invite suppliers to tender for a contract, and may select the supplier who promises to meet his requirements best, perhaps after extended negotiations over which requirements will be satisfied.

Herbert Simon (1996) [1] invented a portmanteau word for satisfying a set of requirements as well as you can: *satisficing* (blending 'satisfying' and 'sufficing'). Industry variously calls this generate-and-select process 'optioneering', 'trade-offs' or 'design pushback'. Table 14.1 illustrates the optioneering challenge on some different types of project.

What all the examples in Table 14.1 have in common is that some of the requirements can't be determined exactly until after optioneering. If you try to be definite in advance, as in:

'The device shall have 4 Gbytes of storage. (Essential)'

then you may have to give way at optioneering time. How do you know that exactly 4 Gbytes of storage is needed? How are you so sure that it is essential? To find out, you need to study the design options, with modelling, simulation or prototyping if necessary. The requirements become clear through design knowledge.

So, how much detail do you need in the requirements before optioneering, and how much afterwards?

- Before optioneering (input), you need enough clarity on stakeholders' goals to know how to compare design options:
 - You should carefully avoid prejudging which option will win. To take a simple example, goals should not refer to 'flash memory' when the storage option is unknown.
 - Most of the goals can will be traded-off in return for other benefits.

Table 14.1: How design options affect requirements.

Project	Design options	Features of these design options (effects on requirements)
Handheld consumer electronics device	Miniature hard disk	Consumes more power Provides more storage Cheaper
	Flash memory only	More reliable (no moving parts) Less sensitive to shocks
'Green' (low environmental impact) car	Carbon-fibre shell	Lighter Less fuel consumed (so, 'greener') Harder to recycle (maybe less 'green')
	Aluminium shell	Lighter than steel Harder to weld
	Steel shell	Easier to manufacture Possibly safer (not flammable)
Retail business software	'Intelligent' personal computers used as clients, minimal server	Uses existing PC network Reliable, easy to replace server Slow to set up temporary networks for a meeting, etc
	'Thin clients' on portable devices, complex server	Allows mobile working Easy to set up anywhere Uses a lot of network bandwidth Server needs major disaster recovery facilities
Transport facilities for new airport terminal	Car park and high-capacity approach roads	Preferred by passengers Generates parking income Increases traffic flows and pollution
	Railway station below terminal building	Fast, reliable, safe, 'green' Costly to build
	Shuttle buses to existing railway station	Quick and cheap to set up Less convenient for passengers (longer journeys)

- After optioneering (**output**), you need to tell your contractors or developers exactly what to deliver, in a measurable form that enables you to verify that you get what you asked for:
 - Since you now know which design option you are going with, you can safely refer to it and to any parts of it (like flash memory) that will certainly be included.

Avoiding the D-word

There is an industry dogma that it is naughty even to mention design until you have finished writing the requirements.

This has a good intention: it is certainly right to find out what your project's goals are before committing to a solution. Jumping to design conclusions, or 'solutioneering' as some call it, can lead to costly mistakes. It is right to spend time discovering the requirements first.

But trying to write product specifications without making any assumptions about design is frankly silly. We have seen people trying to specify the capabilities of a warship without using any words that admit that a solution has been chosen! It goes like this:

'The ship, um, I mean the system, shall be able to sail, um, I mean manoeuvre ...'

If the word 'system' is also forbidden by the thought police, then the language can become really contorted:

'The force projection capability shall include the ability to manoeuvre ...'

But a product or 'system' specification *always* assumes a known design approach. If you know your force projection capability is a ship, call it one. Then you can specify how fast it must sail, simply and directly.

We suspect that part of the trouble is the use of the term 'requirements' for both:

- **input-prioritised pre-optioneering goals**, i.e. results that people want in the world; and
- **output-prioritised post-optioneering specifications**, i.e. design elements chosen for a product.

We may observe that in the traditional phrases 'user requirements' and 'system requirements', the component words are dangerously ambiguous and even obviously wrong. More rational phrases in the spirit of this book might be 'stakeholder goals' and 'product requirements'.

- All the requirements need to be precise enough to support costing, detailed design, and testing.

Optioneering may need to be repeated:

- In an evolutionary life cycle, optioneering may be needed to find the best approach for each (revised) product, as market conditions change.
- In a complex system development, optioneering may need to be conducted to find the best design for each subsystem and sub-subsystem, as you go down through the levels of the hierarchy.

14.2.3 The Optioneering Process

Optioneering is the process of selecting the best available option for your system's design, given your project's input requirements (Figure 14.1). It consists of the following steps, which are often done partly in parallel:

1. Discover and document the input requirements. We shall assume this means newly-discovered goals from your stakeholders, but if you are making a subsystem, it more likely means understanding, interpreting and correcting requirements inherited from the parent system.
2. Identify design options that could meet the input requirements. For example, if yours is a software project, identify alternative software architectures.
3. Translate the input requirements into measurable criteria for evaluating the design options. Well-written requirements should form good criteria. Note, however, that if a requirement will be met perfectly by all the options, it is no use as a criterion for telling the options apart.

COTS Trade-offs

A common experience of having to make trade-offs is when the solution is to consist of a commercial off-the-shelf (COTS) product, like a car, a music player or a software tool. Probably, *none* of the COTS products on the market exactly satisfy your requirements – and if they do, they are very expensive.

The difficulty is compounded when the solution is to assemble two or more COTS products into a system. It will be held together with human-implemented procedures or custom software for exporting data from one package, reformatting it, and importing it into the next package, etc. Here, the trade-offs should take into account the risk introduced by the procedures and custom software.

4. Evaluate the design options against the criteria.
5. From the evaluation results, identify which option is best.

A Natural Divide?

Perhaps optioneering forms a natural divide between 'soft', imprecise stakeholder goals at the start of a project, and precise, measurable product requirements to be imposed on contractors, once the design is known.

Note that people working in some organisations may never see over the divide at all. For example, a software house that develops applications in the context of a large system such as a database may always work in an essentially fixed architectural framework. Large design choices do not then arise.

This would explain the controversy between people who believe that requirements work is always about people, negotiation and compromise (their work is pre-optioneering), and people who insist that requirements must be formally documented, precise and measurable (their work is post-optioneering).

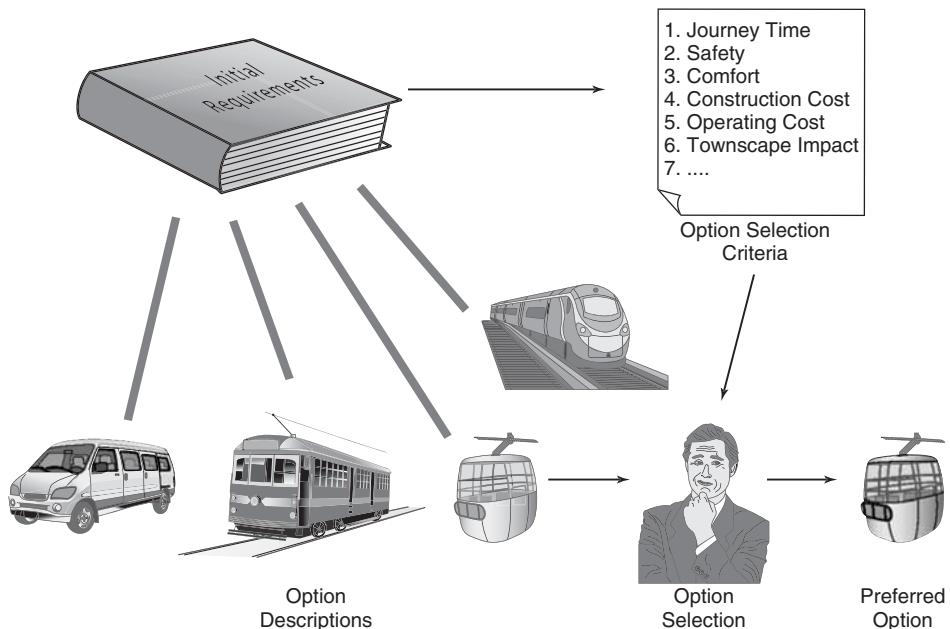


Figure 14.1: The basic optioneering process.

Step 1 is described in the rest of this book. Step 2 is out of our scope. Step 3 is described in Chapter 9. Step 4 depends on the domains involved. We will, therefore, concentrate on step 5: picking the winner.

14.2.4 Selecting the Winning Option

The best is the enemy of the good.

Voltaire

You, or a panel of experts in the different areas involved, score the design options as accurately and objectively as possible on the listed criteria (the input-prioritised goals from the project's stakeholders).

You then have a table of scores, for each option against each criterion. If you are really lucky, it will be obvious at once which option is the winner. That will happen only if almost all the criteria point in the same direction, i.e. if the rankings on all the criteria correlate well.

There are statistical measures such as Kendall or Spearman [7] that you can use to assess correlations of rank (1st, 2nd, 3rd ...). However, that often isn't necessary; if there is a strong correlation, you'll see it at once (for example, option C comes top on 17 out of 20 criteria). Conversely, if there is no strong correlation, it won't help you much to be told that by a statistic.

So you are quite likely to have a practical problem in evaluating options: how can you convert a set of scores into a result? Here are the main methods that people have tried. Some of these are not recommended; we mention them because they are common practice and we want to show there are better methods available.

Most of the methods are quite quick to apply in practice, and good tools are available. If your project decisions are important, the amount of effort taken on decision-making will be tiny compared to the cost of a bad decision (a likely outcome if you take an unscientific approach).

Debate

The first thing to do when you get a set of conflicting opinions on a project issue is to discuss them.

This may lead to a consensus, and a well thought-out rationale for deciding which option is the winner. If this happens on your project, you may want to use one of the techniques for documenting rationale described in Chapter 7 to explain your decision.

If you do not easily reach a consensus, other methods will be needed.

Structure can be placed on debate by modelling its rationale (see Chapter 7) as chains of assumptions for and against a conclusion (i.e. the warrant and the

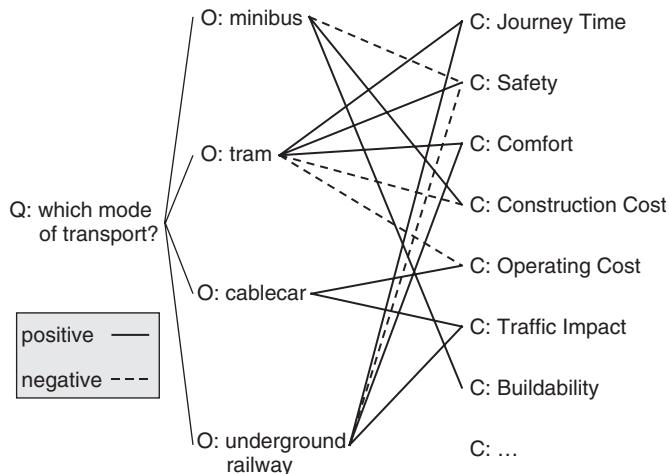


Figure 14.2: Questions, Options, Criteria (QOC) diagram.

rebuttal arguments). A rationale model will often help you to apply one of the following methods.

QOC

A straightforward approach that is suited to trade-off analysis is ‘questions, options, criteria’ (QOC) [13]. Each of these three categories is listed in a column. Solid lines are drawn between them to indicate positive (supporting) connections, and dashed lines for negative (hindering) connections (Figure 14.2). Connections between criteria and options represent assessments. In the language of Chapter 7, this is traceability of alternative options to goals.

Assessments can also be indicated in a criteria/options decision matrix (like Table 14.2 but with just two allowed values per cell). QOC seems to work best when there are not too many criteria to clutter the diagram. You may also find it helpful to show (perhaps by filtering or by changing line colours) the positive and negative relationships separately. This may be sufficient to enable your team to reach consensus.

Weighting Approaches

Projects are often tempted to try to reach a decision by combining scores on different criteria to yield a single final score for each option. The options are then ranked by score to identify the winning option more or less mechanically. We have come to believe that all such approaches are flawed. Let us explain why.

Naïve Weighting: Hoping All Criteria are Equal

The first thing that projects try is simply to add up an option's scores on all the criteria. There are challenges to this:

- Some criteria should lead to the rejection (triaging out) of any option that fails to perform adequately. It does not make sense to give a low score to an option on a triage criterion, and then just to add up that score with the rest. If you did that, a score of zero on safety could wrongly be overridden by high scores on other areas, such as low cost or early delivery.
- Some criteria are more important than others.

Challenges to Weighting

Let us suppose your project triages out clearly failing options on grounds of safety, etc, removing the first challenge. (See Chapter 10, on triage.)

The next step down the weighting road is to try to assign weights to criteria so that scores can be added fairly. There are challenges to this, too:

- It is more difficult to assign weights fairly than people expect. However weights are assigned, the options that some people like may not win. (This could mean that some of those people's requirements have been missed, or simply that people have prejudices.) People can always adjust the weights or the list of criteria to favour certain options. Scoring, too, may be subjective.
- When factors are measured in different units (pounds sterling, tons of Co₂, metres per second, etc) then they cannot fairly be combined – you can't add apples to oranges. Weighting attempts to reduce all criteria to just one dimension. Unfortunately, there is no mathematical basis for doing this: the procedure cannot be justified.
- Why should the relative importance of different criteria be the same in every situation? Isn't the local balance of criteria precisely what we are trying to determine in a particular trade-off?
- Suppose for the sake of argument that you do succeed in finding the perfect set of weights. Suppose also, that most of the criteria point to option A, but some important criteria point to option B, which is radically different. After months of intensive evaluation, option A gets a weighted score of 50.5 and Option B scores 49.5. Are you content to believe that your weights have correctly integrated the factors, or do you want to get the best human brains to study the conflicting criteria?

You can possibly get around the problem of fiddling with weights by keeping an audit trail, using standardised criteria, publishing the weights to be used, etc.

But weighting approaches (no matter how sophisticated: we will mention value engineering and analytic hierarchical process (AHP) as examples) cannot

avoid the apples and oranges challenge. Different criteria, like safety, air pollution, convenience to shoppers and ease of maintenance, are as dissimilar as chalk and cheese. The basic issue is that it is inherently unreasonable to reduce a complex problem to a single number for each option, by any mechanical procedure. Weighting puts many brightly coloured strands into the blender, reducing them to a brownish grey sludge. Meaning is lost.

Value Engineering

There is a strong trend in modern business practice to try to measure everything in money terms. In a business owned by shareholders, directors have a duty to provide a financial return; and all organisations should use their money efficiently. Other things being equal, projects with high benefit and low cost are to be preferred. Those other things include the law, which may force organisations to consider impacts on society and the environment.

There are three groups of factors that can be expressed in money terms:

- value;
- risk;
- cost.

Value is the price that the business puts on having a product or feature. A requirement, such as a functional feature, is worth providing if its value to stakeholders exceeds its cost. Say a feature is estimated to cost £100, but the business estimates its value to customers at £110. The feature is worth implementing as it has a positive net value of £10. Different factors may contribute to value. For example, you can separate the value of a requirement to the business, and the additional customer satisfaction that the requirement brings. 'Value engineering' attempts to maximise value by selecting requirements according to the value they offer.

Risk can be stated in many different ways. A simple one is the contingency felt to be needed in case of overrun. Say the estimated cost of a requirement is £100 and the contingency for it is £30. The total cost is therefore crudely estimated at £130. Taking risk into account, the total cost is greater than the £110 value of the requirement. On this basis, the requirement should not be implemented, unless other factors come into play.

See also the section 'Competing for Resources: Cost-Benefit Analysis' in Chapter 10.

Can everything be priced?

There are many factors, however, that cannot reliably be incorporated into value analysis. Social and environmental benefits (or disbenefits, if a project's impact is negative) are difficult to place a money value on. They can be priced very differently depending on the assumptions made. For example,

what is the environmental cost of producing a tonne of Co₂? It may be zero, if nobody cares. If there is a carbon-trading scheme, then it may be the cost of the needed carbon credits, which varies from day to day with the carbon credit market. If it is the cost of counteracting global warming, the cost may be immense.

Reducing every factor to a price is equivalent to weighting.

There is an alternative view of 'value', which is that the value of a product corresponds to how well stakeholder needs are met. The techniques in this book are designed to help you discover such needs. But you can't add up values of that kind on a spreadsheet.

Analytic Hierarchical Process (AHP)

A sophisticated approach to assigning weights is the analytic hierarchical process (AHP) proposed by Saaty [11]. This asks you to compare items in pairs, and give each pair a relative score. This is applied first to criteria, then to options. For example, if you consider that safety is far more important than comfort, a score of 9 is given to safety, relative to comfort. AHP then calculates relative weights for the criteria. You are then asked to score each option on each criterion. (Compare this to Figure 14.1; you can see that it fits the basic process well.) Finally, AHP calculates a ranking (from best to worst) for the options, thus identifying the winner and runners-up directly.

AHP is used in industry, and may sometimes give good results. However, it has several weaknesses. Like all forms of weighting, AHP is potentially vulnerable to bias. There is no statistical (mathematical) theory underlying AHP; for instance, values like the 9 just mentioned are arbitrary. On a practical note, AHP demands a rapidly growing number of comparisons as the number of criteria grows, so it could be time consuming to apply.

Perhaps the real question you need to ask yourself, with any of these weighting methods, is whether you should rely entirely on a mechanical approach to make important decisions.

Quality Function Deployment (QFD)

Quality function deployment (QFD) is a trade-off method devised by Yoji Akao to relate design elements to stakeholder requirements [12]. It is said to be in wide use in industry, especially in Japan.

QFD is used both for product development (when the design elements are product features) and process improvement (when the design elements are process changes).

QFD typically contains:

- the QFD matrix, relating requirements to design elements;
- a 'House of Quality', interrelating design elements.

QFD Matrix

At the heart of QFD is a matrix of requirements (rows) against design elements (columns). Each cell in the matrix is scored as follows:

- strong (○);
- medium (○);
- weak (▲);
- no effect (cell is left blank).

Negative effects can also be shown.

For example, if one row is ‘useful for outdoor pursuits’, then the cell where that intersects with the column ‘GPS navigation’ is marked ‘strong’ – global positioning is high on the list of useful capabilities (Figure 14.3). Additional matrices, such as the ‘House of Quality’ (explained below) can be added as triangles showing the interactions of various factors.

Requirement attributes such as input priority (see Chapter 10) and rationale (see Chapter 7) can be displayed on the requirement rows to support the trade-off analysis.

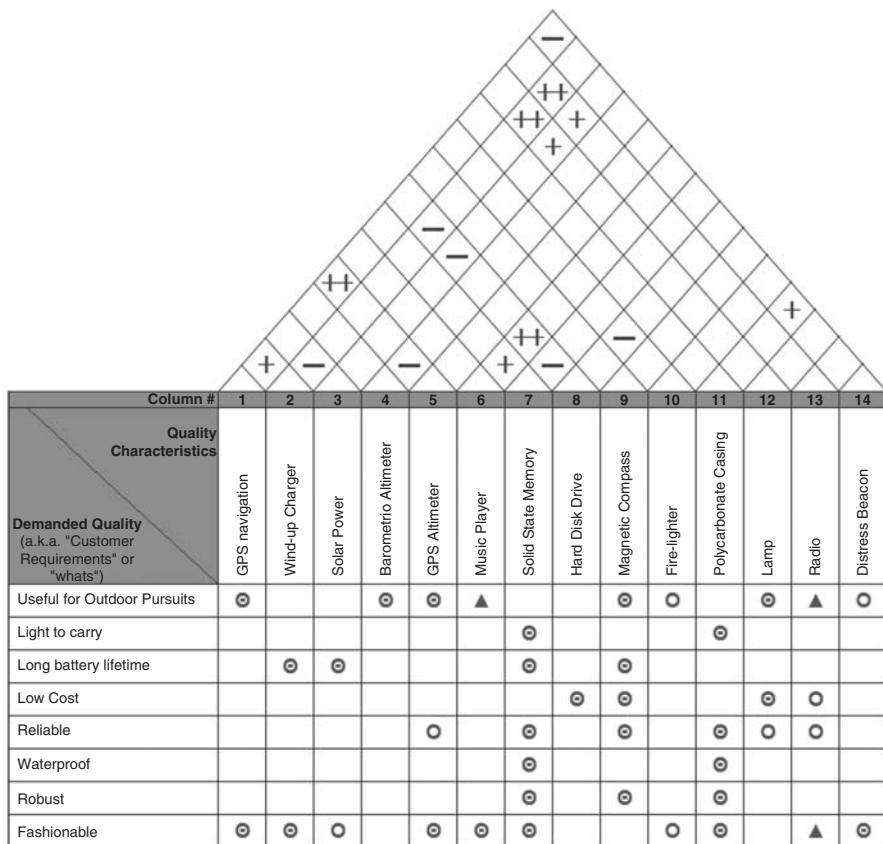


Figure 14.3: QFD matrix with ‘House of Quality’.

Isn't QFD just traditional Traceability?

A systems engineer might say that the QFD matrix is not much more than a 'traceability matrix', as drawn by any requirements database tool, in which a supplier demonstrates how his design complies with or 'satisfies' the customer's requirements, item by item.

Traditional traceability, however, just says: 'Yes, we've satisfied that one by doing this and this'. It's all about ensuring that the chosen design covers each requirement.

QFD differs from a matrix of satisfaction traces in two ways:

1. it allows several competing designs or design elements to be compared;
2. it allows the strength of each trace to be displayed.

The systems engineer is right that you can use a requirements database tool and its traceability mechanism to do QFD, as long as the tool permits you to label a trace (as strong/medium/weak, etc). However, the intention is not to say: 'This completely satisfies that', but to ask: 'Which of these best satisfies that?'.

QFD is basically a qualitative method, able to display the results of much analysis graphically for human judgement. Used qualitatively, it is an excellent tool.

QFD has also encouraged organisations to look at what it calls the 'voice of the customer' (requirements) for the first time – the 'voice of the company' is the design response in the columns of the matrix. Perhaps QFD is requirements thinking, in language acceptable to product management and marketing.

People have inevitably gone further, trying to improve the analysis by adding weights below the matrix to combine different factors. We have already discussed the weaknesses of weighting approaches in this chapter: basically, you can't add apples and oranges.

House of Quality (HoQ)

A frequent but not obligatory addition to QFD is a second matrix, drawn above the main (square) matrix as a triangle to form the pointed roof of the House of Quality as shown in Figure 14.3 [3]. It shows how design elements interact with each other – positively or negatively. Again, you can use symbols to show strong and weak effects.

For example, a '++' symbol in the HoQ triangle means 'these two features go together really well, they help each other strongly'. That fact might influence the choice of features to go in a given release of your product.

Such an ' N^2 ' matrix (each of N features possibly affecting each of the remaining $N-1$ features) is valuable when features can interfere with each

other (see the discussion of feature interaction in Chapter 3). The House of Quality should trigger human discussion and decision-making. QFD with House of Quality is thus quite a good technique for qualitative decision support, at least if the table is small enough to allow you to see the patterns in the data without statistical analysis.

Principal Component Analysis (PCA)

A set of scores on alternative options, measured against a set of criteria, sounds like something that can be analysed mathematically. However, as explained earlier, the apples and oranges nature of criteria that can't be added together seems to invalidate any weighting approach that tries to combine scores on different criteria into one number.

Principal component analysis (PCA) [8] gets around the apples and oranges problem by looking instead for a small number of new dimensions that explain as much of the difference between options as possible. It then leaves decision-making up to you.

PCA may succeed or fail: your situation may produce a clear winner, or it may not:

- **If PCA succeeds**, you get most of the variation between options (say, 75%) explained by the first two or three dimensions that PCA creates.
You can then display each option's position in this new, simplified space (a plane, if there are just two dimensions). Similar options will be close together; dissimilar options will be far apart.
Even better, you can display each criterion as a vector – a line with both a length and a direction – from the centre of the space found by PCA. You can then actually *see* what kind of decision you have to make. For example:
 - **A clear winner**, as one option is favoured by many of the criteria, including the criteria your stakeholders consider most important.
 - **A real dilemma**, as one option is good in one way, but another option is good in another. PCA is telling you gently but firmly that your project needs to face up to the facts and make the best decision it can. The PCA chart will show some options clearly favoured by one group of criteria, and other options clearly favoured by another group of criteria.
 - **The options are so similar** that it hardly matters which you choose. Perhaps you already triaged out (rejected) all the bad options before running the PCA on the few remaining good-looking choices. The PCA chart may show the options scattered about, each favoured slightly by some of the criteria. The criteria may point in all directions like a sunburst, but the difference in scores is small.
- **If PCA fails**, the first few dimensions only explain a small part of the variation between your options. Essentially, PCA is telling you that you

have a hard decision to make. There is no clear pattern to the data. This could happen when all the options are good, for example.

We have found PCA to be a helpful tool in optioneering. It is not a magic bullet: it does not work every time and even when it does, the decision remains yours to take.

14.2.5 Optioneering with PCA: A Worked Example

This example has two parts:

1. **Optioneering problem, criteria and scoring.** This part is independent of the approach you choose for interpreting the results and arriving at a winner.
2. **Decision-making with PCA.** This shows how PCA can simplify your decision without taking away human responsibility for it.

Optioneering Problem, Criteria and Scoring

Anytown has a traffic problem, which is growing rapidly worse. The transport authority wants to provide a quick, safe, comfortable, affordable and reliable link between its Main Street shopping area and the railway station. The town is on hilly ground and the roads are choked with cars. The authority would like to minimise any negative impact on road traffic, and ideally would like traffic to decrease as people change their mode of transport from car to public transport. The solution has to be affordable and realistic to build and operate. Finally, they would like to avoid damage to townscape and wildlife.

Your team decides to carry out an optioneering study to pick the best solution (Figure 14.1) to meet the transport authority's requirements.

Your team identifies four options:

1. A cheap minibus shuttle on existing roads. This can be set up quickly, but it won't cut journey time much, or do much to decrease road traffic.
2. A tram running on rails in existing roads. This will cost more to build, but will run much faster. It will probably make traffic jams worse.
3. A light railway running in a new tunnel. This will be very expensive, but very fast and reliable. The fire department considers transport in tunnels to be a safety hazard.
4. A cablecar running directly overhead on pylons. This will completely avoid traffic and be economical to run once installed. It will have a high capacity, as cars arrive every few minutes.

You identify 11 criteria (the column on the left of Table 14.2) that you believe reflect the key qualities that any acceptable solution must have. You also expect

Table 14.2: Scores awarded on qualitative seven-point scale from --- to +++

Options		Minibus	Tram	Railway	Cablecar
Criteria					
Journey_Time	0	+++	+++	-	
Safety	---	+++	--	0	
Comfort	+	++	+++	--	
Construction_Cost	+++	--	---	-	
Operating_Cost	+	--	---	++	
Townscape	-	--	0	0	
Wildlife	-	0	0	-	
Traffic_Impact	-	---	+++	+++	
Buildability	+++	-	-	+	
Capacity	---	++	+++	+++	
Reliability	--	++	+++	++	

Choosing Measurement Scales

There is nothing special about a seven-point scale, except that it's a good match to the human ability to deal with things qualitatively.

Measurements should be quantitative where a criterion can be calculated or predicted accurately, e.g. journey time reduction can be measured in minutes.

Qualitative criteria should be used where appropriate. Qualitative does not mean vague and subjective. You should define what '+++' and ' - ' mean for each of your measurement scales, so that the scoring is fair and repeatable.

You should make all your measurement scales run in the same direction, e.g. a higher number is always better.

these to be the points on which the solutions will differ: if all of them score the same on a particular criterion (such as whether they run from Main Street to the station), that criterion is no use in telling the options apart, however important it is.

Your team of transport, environmental and other experts scores each option in an appropriate way, quantitatively where possible. The results are then transferred to a seven-point relative scale: ' - ' means a strong negative impact, '0' means neutral, and '+++' means a strong positive impact. For example, costs are estimated from the option designs by consulting engineers, and translated into values on the seven-point scale. Capacity is calculated as the number of vehicles per hour times vehicle capacity, and again translated onto the scale. The results are shown in Table 14.2.

Table 14.3: Scores transposed to 1 ... 7 scale ready for statistical analysis.

	Journey-Time	Safety	Comfort	Construction_Cost	Operating_Cost	Townscape	Wildlife	Traffic_Impact	Buildability	Capacity	Reliability
Minibus	4	1	5	7	5	3	3	3	7	1	2
Tram	7	7	6	2	2	2	4	1	3	6	6
Railway	7	2	7	1	1	4	4	7	1	7	7
Cablecar	3	4	2	3	6	4	3	7	5	7	6

You could stop here, and try to decide on the basis of this evidence which option is best. You could, for instance, argue that the minibus option should be triaged out (rejected before detailed evaluation) as it is much less safe than the other options. In fact, if it falls below the minimum acceptable degree of safety, it must be triaged out. You could then try to judge which of the remaining options to prefer. The railway has the most '+++' scores, but is that a defensible basis for a decision?

Let us instead look at how far PCA can help us decide what to do.

Decision-making with PCA

This part of the worked example shows how a particular statistical method, PCA, can support but not replace good human decision-making.

The scores of Table 14.2 are transposed without change in meaning to a form that a statistics package can accept (Table 14.3): '-' is replaced with 1, '+++' with 7, and so on.

The statistics package is used to run a PCA. In essence, what PCA does is to take the many dimensions of an evaluation – like the 11 criteria (column names) in Table 14.3 – and try to explain as much as possible of the difference between the options (row names) with just a few new dimensions. Mathematicians call these 'eigenvectors'; in PCA, the largest of these are called 'principal components'.

The hope is that the analysis will show the differences in a clear and simple way that suggests two things:

- **an interpretation**, e.g. that traffic impact and cost are the main factors separating your options;
- **the grounds for your decision**, e.g. that if you mainly care about traffic impact, you will prefer one of these options here, whereas if cost is your concern, you will prefer one of those options over there.

Table 14.4: Summary of PCA.

Principal component	Eigenvalue	% of variance explained	
1	15.16	56.08	Using Correlation Matrix,
2	7.59	28.05	Singular Value Decomposition
3	4.29	15.87	Jolliffe cut-off 1.72
4	0.00	0.00	
5	0.00	0.00	

If PCA works well on your data, just two or three principal components will explain most (say, 80%) of the variation in scores between the options.

Conversely, if it takes five or more principal components to explain half the variance, PCA has essentially failed: it has not found a simple clear message in the data.

Table 14.4 shows that PCA has worked well on Anytown's transport trade-off. It shows that from the 11 criteria, all the variance can be explained with just three new dimensions (principal components).

A graph of this is called a 'scree plot' (Figure 14.4). In this case, we have a nice sharp result – a slope as steep as the loose rocky screes that drop down glaciated slopes, hence the name of the plot. If the scree plot instead shows a gentle slope, PCA can't help you.

You can also look at a rough and ready statistic called the Jolliffe cut-off. This tries to suggest how many principal components you should consider. The value here (1.72) implies that, probably, only the first two dimensions should be considered, so we can display both of them on a plane as the X and Y axes (Figure 14.5). You can see this anyway from the '% of variance explained' in Table 14.4: the first two dimensions explain over 84% of the differences between the options. This is a nice, sharp result.

Some statistics packages offer various refinements on basic PCA. The results in the worked example here were calculated using singular value decomposition (SVD), which sharpens the results, leading to larger eigenvalues for the first few principal components. A correlation matrix (rather than a variance-covariance matrix) is the correct type for this kind of problem. For a more technical account of PCA, see [8].

The options are shown as the corners of the polygon; the criteria, which started life as separate dimensions, have all been projected or flattened on to the plane of the paper defined by the first two components.

It should at once be clear from Figure 14.5 that the criteria conflict, i.e. they pull in different directions:

- If you care mainly about costs, then you will choose the cablecar or the minibus.

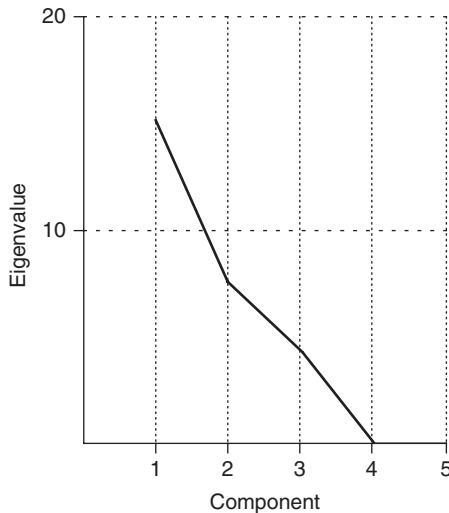


Figure 14.4: A scree plot showing that PCA has succeeded in reducing the number of dimensions.

PCA in Theory and Practice

Optionering with PCA may seem difficult at first sight, and indeed it takes some pages to explain. The technique is in practice quick to apply, once you understand it.

The technique is illustrated here with a small example. In practice, you would:

- use a larger number of criteria (perhaps 30–40);
- use experts to evaluate the options, quantitatively where possible, against the criteria in their areas of expertise;
- derive the individual scores (+++ etc) from experts' findings.

The mathematical approach is unaffected by these differences.

- If you care mainly about capacity, reliability and effect on road traffic, you will go for the cablecar or the railway in its tunnel.
- If comfort and journey time are your main concerns, while traffic is of no interest to you, you should choose the tram.
- If you want a fast, comfortable, reliable, high-capacity system and don't care about cost, you will certainly prefer the railway.

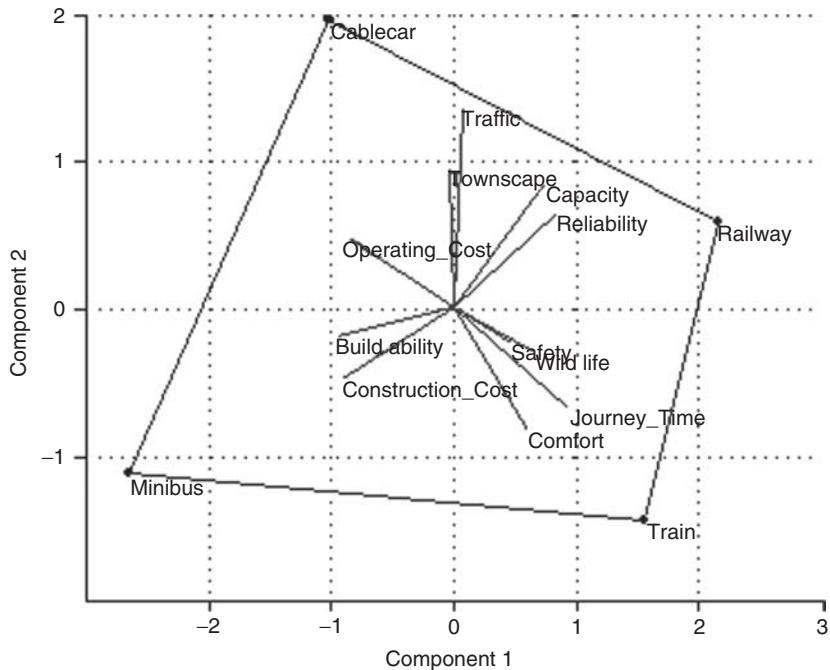


Figure 14.5: Principal components analysis, biplot of criteria on option chart.

Tips for Optioneering

- Start with a clear understanding of your stakeholders' goals.
- Make sure you know and have accurately documented any triage constraints, i.e. those that cannot be traded-off.
- In your search for possible options, make sure you spread the net widely enough. If you fail to consider a feasible approach, you might have to start the whole exercise again.
- Be ready to 'triage out' an option as soon as it is clear that it is entirely unsuitable, but document carefully the evidence for dropping it.
- Weighting is subjective; there is no fair way to choose weights, and it is dangerously easy for someone to choose weights that give the result they want. There is no mathematical basis for reducing all criteria to one dimension (such as value or merit), which is what weighting does.
- 'If you want honey, try bees.' If you need to do more statistics than you are comfortable with, get a statistician to advise your team.
- Not everything can be expressed in terms of money. Risk and customer satisfaction (value in a nonfinancial sense) are always important.

Which should you choose? The statistics quite rightly do not tell you; that depends on how you feel the different criteria should be balanced out in this case.

Anytown council has a strong concern for traffic, townscape, reliability, capacity and cost, outweighing other factors. Component 2 correlates positively with impact on traffic, townscape, capacity and reliability, but negatively with comfort and journey time. This suggests that the most suitable options are high on Component 2 (the y-axis of Figure 14.5).

Component 1 correlates negatively with construction and operating costs, so the most suitable options, in Anytown council's view, are low on Component 1 (the x-axis of Figure 14.5). Recalling from Table 14.4 that Component 1 is nearly twice as significant as Component 2 in telling the options apart (53% of the variance, compared to 30%), it is clear that cost is a major factor here.

Notice that there is no implication from PCA that options high on Components 1 and 2 are necessarily 'good' – correlations of components with criteria can be positive or negative. The statistics simply promise to spread the options out as much as possible, not in any particular direction.

The council looks carefully at the cost of the railway but, in the absence of outside sources of funding, concludes that it cannot fund that option and must therefore triage it out. The council therefore decides on the only remaining option in the most suitable group, the cablecar.

There is nothing to stop the council exploring alternative cablecar options, i.e. carrying out another, finer-grained round of optioneering. There are many types of cablecar, varying widely in cost and capacity. It may well be that an improved solution can be found by studying cablecar options in more detail.

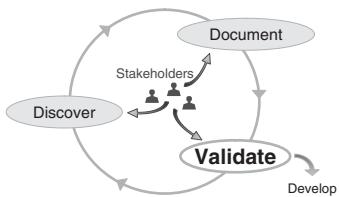
What Requirement Elements do you Discover from Optioneering?

Optioneering is plainly not the best time to find yourself stumbling over stakeholders' goals that you somehow missed earlier.

On the other hand, defining (for the first time) the list of goals that will actually be implemented, giving a rationale for the decisions on the whole product or system, and agreeing measurements and priorities for the accepted goals, is certainly 'discovery'.

So, seen as a discovery context, optioneering is rarely a source of new goals or interfaces (but see Table 15.3 in Chapter 15 for a special case). It is, though, a uniquely powerful discovery context for several essential requirement elements.

14.3 Validating your Trade-offs



As emphasised in this chapter, the ultimate authority for trade-off decisions is human. You can't do better than get the best informed judgement on the available evidence.

Therefore, if your trade-offs are based on evidence which remains valid, and it has been properly considered, the decision must be considered valid. You need only check that you have documented it correctly – e.g. by asking trade-off workshop participants to review the workshop report.

If your trade-offs are based on a method (presumably involving weighting) which has not involved human decision-making at each point, then you must have the results validated by appropriately knowledgeable people who understand the assumptions that have been made in the calculations. Any automated findings must be treated as suspect until they have been properly validated.

14.4 The Bare Minimum of Trade-offs

- None, if you are lucky, and you have a well-defined context.
- Make sure you know you are getting the best available solution, given:
 - possibly conflicting goals of the project;
 - alternative solutions with different merits.

14.5 Next Steps

Once you know which design option your project has chosen, you can tell which requirements must be rejected because they are not possible in that design. You can set their output priorities (see Chapter 10) to 'Never'.

Other requirements will need to have their measurements (e.g. acceptance criteria) adjusted to suit the chosen design (see Chapter 9).

Your project now needs to decide whether to implement the remaining requirements at once, or in which future phase: again, a matter of output priorities.

That done, the requirements are now basically ready to drive all remaining development activities, such as software development, testing and documentation. Chapter 15 discusses some remaining issues in transforming the requirements into a project roadmap.

14.6 Exercises

You have achieved a major goal in your life (promotion, marriage, a house with a garden, a conference you've organised, etc). You want to celebrate it with a really nice meal for your friends.

Your **requirements** are that:

- the surroundings must be elegant;
- the food should be original and delicious;
- the service must be efficient, professional and unobtrusive;
- you have a generous but not unlimited budget for the event;
- you don't want to risk anything going wrong on the day.

The **options** you have identified are:

- a chic and very beautiful French restaurant, not yet too expensive, with a young chef who is quickly gaining a reputation;
 - a big, traditional hotel known for its attractive rooms and gardens, solidly reliable service and good, plain food;
 - a personal catering service that brings a cook and a team of waiters to your home; the cook comes to plan the meal in advance, checks your kitchen and dining room, and discusses the menu with you;
 - organising it yourself, choosing wines and food, and hiring a couple of catering staff to act as cooks/waiters for the weekend (to prepare everything the day before and to serve on the day).
- a. Draw a matrix of options (rows) against requirements (columns). Score each option on each requirement using a seven-point scale from '–' (strongly unsatisfactory) to '+++' (strongly satisfactory).
 - b. Identify any 'show-stoppers' with a bold cross through affected cells in the matrix. Remove the affected options from consideration.
 - c. Write a reasoned argument (modelling the rationale if need be) for your final choice.

14.7 Further Reading

14.7.1 Trade-offs

1. Simon, H.A. (1996) *The Sciences of the Artificial*, 3rd Edition, Cambridge, Mass: MIT Press.

Herbert Simon is the father of trade-off analysis. He was the first person to describe clearly the problem of satisfying a set of needs as fully as possible, given an imperfect set of design options.

Few requirements books cover the process of trading-off requirements against design at all, perhaps because requirements work is thought to finish when design starts. The following are some books that do cover the subject.

2. Young, R.R. (2001) *Effective Requirements Practices*, Boston: Addison-Wesley.

Ralph Young documents a large number of requirement practices suitable for complex development projects. His book is very clear on the need for trade-offs.

3. Stevens, R., Brook, P., Jackson, K. and Arnold, S. (1998) *Systems Engineering, Coping with Complexity*, London: Prentice Hall.

Stevens *et al* mention trade-offs, suggesting cost-benefit analysis and QFD as methods.

4. Maier, M.W. and Rechtin, E. (2000) *The Art of Systems Architecting*, Boca Raton: CRC Press.

Maier and Rechtin write for the designers of large systems such as spacecraft (Rechtin was chief architect to NASA for the Apollo moon landings). This is 'hard' systems thinking, at the opposite end of the scale from Peter Checkland's *Soft Systems Methodology*. The book identifies principles and rules of thumb for design.

5. Rechtin, E. (1990) *Systems Architecting, Creating and Building Complex Systems*, Upper Saddle River, NJ: Prentice-Hall.

Maier and Rechtin's book is based on this, which was perhaps a better book.

6. Parnas, D.L. and Clements, P.C. (2001) A Rational Design Process: How and Why to Fake IT. In Hoffman, D.M. and Weiss, D.M. *Software Fundamentals, collected papers by David L. Parnas*, Boston: Addison-Wesley.

David Parnas has pioneered several aspects of software development, including the relationship between requirements and design.

14.7.2 Statistics

7. Dytham, C. (2003) *Choosing and Using Statistics*, 2nd Edition, Malden: Blackwell.
A good introductory text.

14.7.3 PCA

8. Smith, L.I. (2002) *A Tutorial on Principal Components Analysis*, available at: http://csnet.otago.ac.nz/cosc453/student_tutorials/principal-components.pdf.

Descriptions of PCA in statistics textbooks are often dry, and hard to apply to optioneering. This, however, is rather a good mathematical presentation.

14.7.4 Weighting Approaches

9. Wiegers, K.E. (2003) *Software Requirements*, 2nd Edition, Redmond: Microsoft Press.
Karl Wiegers proposes using weights on value, cost and risk to calculate an output priority for each requirement.
10. Robertson, S. and Robertson, J. (2006) *Mastering the Requirements Process*, 2nd Edition, Upper Saddle River, NJ: Addison-Wesley.
The Robertsons similarly propose a spreadsheet that calculates output priorities based on weighted value, cost, and ease of implementation (i.e. risk).

14.7.5 Analytic Hierarchy Process (AHP)

11. A good basic description of the analytic hierarchy process is available on Wikipedia at http://en.wikipedia.org/wiki/Analytic_hierarchy_process

14.7.6 Quality Function Deployment (QFD)

An example of QFD with House of Quality is given in Stevens *et al's Systems Engineering* [3].

12. A general account and worked example of QFD is given in Wikipedia at http://en.wikipedia.org/wiki/Quality_function_deployment

14.7.7 Questions, Options, Criteria (QOC)

13. MacLean, A. and McKerlie, D. (1995) Design Space Analysis and Use Representations. In Carroll, J.M. (Ed) *Scenario-Based Design: Envisioning Work and Technology in System Development*, New York: John Wiley & Sons, Inc.
A readable account of QOC, applied to trading-off user interface designs.

CHAPTER FIFTEEN

Putting it all Together

The whole is more than the sum of its parts.

Aristotle, *Metaphysica*

Requirement Elements	Priorities	Measurements	Definitions	Rationale and Assumptions	Qualities and Constraints	Scenarios	Context, Interfaces, Scope	Goals	Stakeholders
Discovery Contexts									
Introduction									
From Individuals									
From Groups									
From Things									
Trade-Offs									
Putting it all Together									

Answering the questions:

- How do all these concepts and techniques fit together?
- Do you have to use them on every project?
- How should all this information be organised to put your project on the right track?
 - ... so the product's requirements are fully understood
 - ... so as not to waste effort
 - ... so the project succeeds.

15.1 Summary

Almost every project activity depends on requirements. But projects have widely differing contexts, and need their requirements structured appropriately.

The chapter looks first at how to use the matrix of discovery contexts and requirement elements to design the right process for your project, with case studies, and how to use requirements to drive project management.

The chapter then looks at how to organise requirements. The value and limitations of use cases are discussed, along with ways of organising product functions. A list of traditional 'shall' statements has its place, but does not address the need to relate requirements of different types or the conflicting needs of different readers of requirements. These pressures push projects towards the use of requirements tools that automate traceability between requirements and provide alternative views of project information.

The view that individual requirement elements can be used on their own to discover and organise requirements is discussed. The chapter concludes that a more balanced approach, using a combination of requirement elements tailored to your project, is likely to be best.

15.2 After Discovery

15.2.1 Everything Depends on the Requirements

When you have found out the requirements for your project, your work is just beginning. You must:

- organise the requirements for project use, for example in a traceability database tool;
- plan how to implement the requirements;
- design according to the requirements;
- monitor and control the project to ensure the requirements are being implemented;

- anticipate changes in the requirements, and ensure that they don't jeopardize the success of the project;
- verify (usually by test) that the requirements have been met;
- document the use and maintenance of the product (i.e. its operational scenarios) with manuals and training materials;
- support the embedding of the new capabilities in the organisation.

Strikingly, all of these activities critically depend on the requirements: on discovering them, and then on managing the project from them.

The many uses of requirements place tight constraints on how the requirements themselves are organised. Indeed, some of these constraints may seem to conflict; should you keep qualities with the functions to which they apply, or all together in a separate quality section? There are reasons for doing both.

So, at the risk of straying beyond the strict scope of requirements discovery, this chapter looks at how the elements you have discovered – goals, scenarios and so on – fit together to guide the rest of the work on your project.

15.2.2 Principles for the Requirements Chef

No two projects are alike. What impact does that have on your requirements, and how to organise them? This chapter does not try to say exactly what process recipe you should provide for your project. What it does instead is to

Avoiding Process Bias

Personal process bias is a constant danger. It is easy to become enthusiastic about the latest method (agile, use cases or aspects, perhaps) or to swear that a familiar method (context diagrams, dataflows and statecharts, maybe) is really the right way to approach things.

The best defence against bias is to be aware that it exists. After that, there are several ways to minimise bias:

- learn a wide range of complementary techniques, so as to view each situation from different angles;
- reflect on the implications of the tables in this chapter: that projects are all different and need processes to be customised just for them;
- consult as wide a range of stakeholders as possible;
- get the whole team involved in writing and reviewing the requirements.

Each individual opinion may well be biased in some way. But, as in a theatre, the separate red, green and blue lamps should end up bathing the scene in something approaching pure white light.

describe the principles for combining ingredients that you should be following when you create your own requirements process, by looking at:

- **the right process for your project:** what mixes of requirement elements should there be on different types of project?;
- **organising the requirements specification:** how should you present your requirements for your project to use? Do use cases cover everything? Should requirements for systems be handled just like those for software?

These questions are related. They address the tremendous diversity of requirements work that we are just starting to understand.

Arguably, trends like globalisation and outsourcing of development work to offshore companies tend to increase the importance of requirements discovery. Certainly, these trends make communication and explicitness of assumptions and definitions more necessary than ever before.

15.3 The Right Process for your Project

Processes come before Methods.

Methods come before Tools.

Richard Stevens

Different projects contain very different mixes of requirement elements (such as goals and scenarios) as described in Part I of this book. Hence, projects involve very different combinations of the discovery contexts (such as interviewing and prototyping) described in Part II of this book (Table 15.1).

Any one model reflects only a tiny piece of reality, and that only from one point of view. To construct a reasonable picture of reality, you need to put together many interlocking models, many interwoven requirement elements.

Your project is unique. Use this book to identify the right mix of requirement elements and discovery contexts to create the right requirements process for your project's situation.

That process is a part of your development life cycle. Depending on the degree of risk involved, the life cycle may include activities such as concept study, evaluation of alternative designs, and trials before introduction into service. If you have not already identified the right kind of life cycle for your project, see a book such as Richard Stevens' (1998) *Systems Engineering* [1] or Andrew Farncombe's (2004) chapter on tailoring life cycles in *Scenarios, Stories, Use Cases* [2].

There are many types of development projects. Whatever your situation, you will need to discover a mix of different requirement elements from Part I. You can use discovery contexts from Part II for any of those elements, yielding a matrix of possible context/element approaches (Table 15.5 on page 383).

Table 15.1: Examples of dominant requirements elements and discovery methods for different types of products.

Type of product under development with examples	Dominant requirement elements with secondary elements (mainly Part I chapters in parentheses)	Likely discovery contexts (Part II chapters in parentheses)
Financial software e.g. banking, insurance, retail software (central database server, point-of-sale clients)	Scenarios (5), user interface prototypes (13) with security and regulatory requirements	Interviews (11) Observation (11) Workshops (12) Prototyping (13) Archaeology (13)
Real-time and embedded systems e.g. automotive control systems, mobile devices, process control systems ¹ for production lines and power plants	Events and timing constraints (4), event handling scenarios (5), interfaces (4) with safety, security, reliability and other qualities	Templates and Standards (6) Interviews (11) Observation (11) Workshops (12) Prototyping (13) Trade-offs (14)
Planning/concept definition choosing options, e.g. among candidate new technologies; transport planning	Goals (3), assumptions (7) leading to criteria that can be traded-off: possibly few genuinely fixed constraints	Interviews (11) Workshops (12) Trade-offs (14)
Interactive software e.g. graphics editors, spreadsheets, games	Scenarios (5), user interface prototypes (13) with performance targets	Interviews (11) Observation (11) Workshops (12) Prototyping (13)
Scientific software e.g. weather/climate modelling, large simulations	Tables (4) for rules (such as algorithms and equations) with performance, accuracy and other qualities	Templates and Standards (6) Interviews (11) Prototyping (13) Archaeology (13)

15.3.1 Case Study: A Retail IT Project

A large retail organisation was moving from a local to a centralised model of information technology (IT) for economic efficiency. The IT contractor was to take on the role of central service provider, including managing the all-important data, as well as developing and maintaining the software. The key design decision – to centralise – was taken before the contract was signed. The contract fixed the price; the requirements were defined after signature. Effectively, the contractor had agreed to implement whatever the requirements

¹i.e. SCADA (Supervisory Control and Data Acquisition) systems

team specified: the team included members from the contractor's side, who were very familiar with the previous system. The implication for the customer was that anything omitted from the requirements would not be implemented.

Let us step through the requirement elements of Part I of this book to see how they applied to that project:

- There were a small number of important **goals** (Chapter 3) with wide-ranging effects, so a single goal model was constructed. Traces to these goals were left implicit, since, effectively, every use case and requirement would trace to the same few goals.
- The **stakeholders** and **context** (Chapters 2 and 4) were largely unchanged by centralisation; for instance, the payment interfaces to the banks and credit card agencies were unchanged. These were briefly summarised; the software contractor reused the interface code directly.
- Retail operations consisted of a set of functions shared by all retail outlets. These were principally for transactions, which were obvious **use cases** (Chapter 5). Most of these were derived from the behaviour of the previous system via a training simulator and, more importantly, experienced retail staff retrained as use case analysts for the project. There were also some less obvious scenarios, such as for maintenance, audit and backroom data operations.
- Centralisation powerfully affects some key qualities (Chapter 6) of an IT system, including **performance**, **security**, and **availability**. For instance, having a central server introduces a round-trip delay, while opening up the possibility of organisation-wide security breaches and interruption to service. Each of these critical qualities was analysed separately. In addition, the opportunity was taken to explore improvements in **usability**.
- Where the retail scenarios were broadly unchanged (barring usability improvements), rationale (Chapter 7) was not important, but when **issues** did arise, these were listed in a shared data area for management **decision**, and traced to the affected requirements or use cases. Such issue/decision pairs formed an immediate justification for changes.
- Definitions (Chapter 8) formed a significant portion of the work. The main elements here were typical retail **data** for customers, products and payments; **roles** and responsibilities for staff; and a **glossary** of designations.
- Measurements (Chapter 9) involved both suitable **acceptance criteria** (success and failure guarantees) for use cases to be verified on delivery by test, and **quality of service (QoS) measures** for quality requirements, to be verified by regular measurements of achieved performance.
- **Priorities** and **trade-offs** (Chapters 10 and 14) were assessed prior to contract and, as such, were out of the scope of the project itself. New requirements continued to drift in, and were queued in a waiting list.

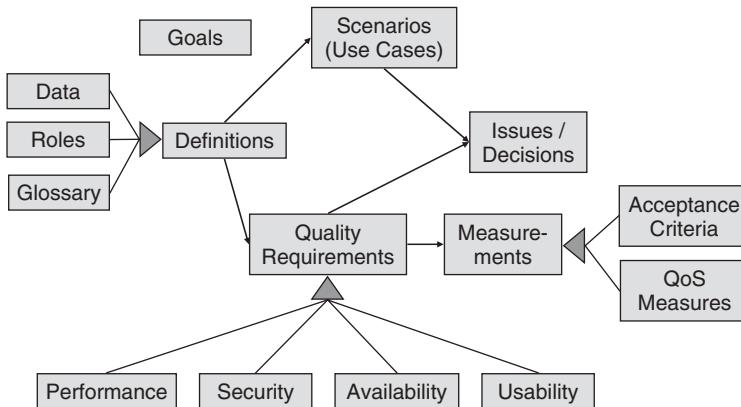


Figure 15.1: Information model for a retail project.

These requirement elements led to the project-specific information model shown in Figure 15.1, along with a requirements process specially tailored to discover, document and validate exactly that set of elements. The main part of that process concerned use cases. These were first documented and reviewed as sketches, then as happy-day scenarios, then as fully dressed use cases complete with exceptions and postconditions.

All of this information was edited and held in a requirements database, lightly customised to provide progress reports and to export into the organisation's document format. (There was much other information in the heads of the experienced developers.)

In terms of the matrix of requirement elements and discovery contexts used on the first page of each chapter of this book, the project had quite a sparsely populated matrix (Table 15.2). In particular, the key trade-off decisions, such as between running costs and both security and performance, had already been made, so the trade-off row is blank. Clearly, other projects may differ widely from this one.

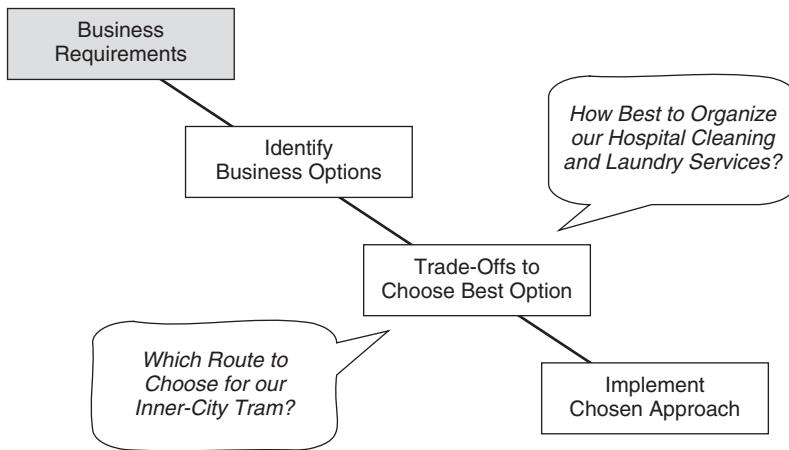
15.3.2 Case Study: Transport Planning

To give a contrasting example of tailoring the requirements process for a project, consider a transport planning project as described in the worked example 'Tram Goals and Trade-offs' in Section 3.2.3 of Chapter 3, and revisited under a different guise in 'Decision-making with PCA' in Chapter 14.

In contrast to the retail project just analysed, scenarios and data definitions are quite unimportant in transport planning. The basic scenarios are almost the same, regardless of the route or mode of transport chosen. The choice of route is not affected by issues of data-handling either, even if (as is likely) some subsystems eventually contain electronics and software.

Table 15.2: Elements/contexts matrix for a retail project.

<i>Requirement Elements</i>	<i>Discovery Contexts</i>	<i>From Individuals</i>	<i>From Groups</i>	<i>From Things</i>	<i>Trade-Offs</i>	<i>Priorities</i>
<i>Measurements</i>						
<i>Definitions</i>						
<i>Rationale and Assumptions</i>						
<i>Qualities and Constraints</i>						
<i>Scenarios</i>						
<i>Context, Interfaces, Scope</i>						
<i>Goals</i>						
<i>Stakeholders</i>						

**Figure 15.2:** Not all requirements processes involve software development.

Instead, the key issues for transport planning are: to identify the stakeholders and their points of view; to prioritise the stakeholders' goals; to identify suitable candidate routes (i.e. outlines of possible design solutions); and to match these together as well as possible, i.e. to conduct trade-offs. Decisions are made against the constant background of the many applicable standards and regulations that impose qualities (such as safety and security) and constraints (such as road widths) on every transport scheme. This is an example of a requirements process that applies far more widely than just to system or software development (Figure 15.2).

Table 15.3: Elements/contexts matrix for a transport planning project.

<i>Requirement Elements</i>						Priorities
<i>Discovery Contexts</i>						Measurements
	Definitions					
From Individuals						
From Groups						
From Things					Qualities and Constraints	
Trade-Offs						Rationale and Assumptions
	Scenarios					
	Context, Interfaces, Scope					
	Goals					
	Stakeholders					

The situation is complicated by the fact that varying the route changes the set of stakeholders affected, as well as the interfaces and scope of the project. For example, if the route runs by a railway station, it will interchange with it – but only if that route is acceptable. Therefore, the project's work consists of a large and somewhat indeterminate set of 'what if' questions, which have to be explored as possible route designs and traded-off against project goals.

The resulting process therefore makes use of a highly distinctive subset of cells in the requirements element/discovery context matrix (Table 15.3). In particular, and unusually, the geographically-local goals and interfaces that can be met are selected somewhat opportunistically according to the possibilities offered by different route options.

15.3.3 Requirements-Driven Project Management

This book has described the work of discovering requirements. That is just the first step to a successful product. Your project must monitor itself – its practices and work products – to ensure that it is on track to implement its requirements correctly, and take appropriate controlling action. So, project management must be driven by requirements.

Some simple examples of the close relationship between requirement elements and management actions are illustrated in Table 15.4. Notice that traditional (contractual) requirements are only one element of this relationship. Risk management, for instance, is driven by identification of many different kinds of risk to the project, derived from a variety of requirement

Table 15.4: Project management activities driven by requirements.

Requirement elements	Project management activities	Transformations needed
Stakeholder analysis	Stakeholder management	Identify importance and attitude of each stakeholder
Goals Assumptions Qualities and constraints	Risk management	Identify unresolved goal conflicts; invert assumptions which are load-bearing and vulnerable; identify qualities and constraints that could be difficult to achieve
Scenarios (User interface design) (System design) (Tests) (User and maintenance manuals) (Training materials)	Planning, monitoring, control, reporting	Identify all the activities (such as design, test and documentation) that the project must perform to implement all the scenarios; construct a plan for completing these activities
Trade-offs Prioritisation	Tender/proposal evaluation, selection of suppliers	Analyse how well each supplier's proposal meets the project's priorities
Measurements, i.e. contractual requirements	Supplier and contract management	Verify (e.g. by test) that the supplier has met all the terms of the contract, including all the itemised requirements

elements: conflicting goals, assumptions that might break, and qualities that might be difficult to achieve.

Approaches that we have found most frequently useful are **highlighted** in Table 15.5; where the intent of an approach can be written as a question, that question is printed *in italics*. It is very likely that your experiences will differ from this.

Taken together, Table 15.1 and Table 15.5 suggest plausible but still rather generalised approaches for various types of project. One day, it may become possible to say: 'On such-and-such a type of project, you will need to use such-and-such a combination of requirement approaches'. Perhaps these tables, sketchy as they are, mark some progress towards making requirements work less of 'a black art' and more of an engineering discipline.

You can find out more about how to translate requirements into management action from Ralph Young's (2001) *Effective Requirements Practices* [3],

Table 15.5: Possible discovery context/requirement element approaches

Requirement Elements	Stakeholders (Chapter 2)	Goals (Chapter 5)	Context, interfaces, scope (Chapter 4)	Scenarios (Chapter 5)	Qualities and constraints (NFRs) (Chapter 6)	Rationale and assumptions (Chapter 7)	Definitions (Chapter 8)	Measurements (Chapter 9)	Priorities (Chapter 10)
Discovery Contexts									
From individuals (Chapter 11)	Who else should we interview about this?	What do you want to be able to do with this?	What interfaces does the product need?	Could you show/teach me how you do task XYZ?	Which qualities are most important in this product?	(Why do you want that?) But ideally, let explanations arise naturally in interviews	What does the term 'XYZ' mean?	How can we measure that? How can we show we have got what we asked for?	Which features of the system are most important? (judged on which criteria?)
From groups (Chapter 12)	Stakeholder (discovery, management) workshop	Goals workshop Threats workshop	Context and scope workshop	Scenarios workshop Exceptions workshop	NFRs workshop	Requirements rationale	Dictionary task in any workshop	Acceptance criteria workshop	Prioritisation workshop Trade-offs workshop
From prototypes (Chapter 13)	—	Experience of missed goals: 'Why doesn't it also ... ?'	Experience of missed events that need to be handled	Experience of workable ways to achieve goals	Experience of qualities that matter most (e.g. speed)	Experience of costs and benefits	—	Expected/attainable performance	Experience of what features people like most, or ask for most
From archaeology (Chapter 13)	Which roles in old system?	What did user guide say you could do?	What interfaces in old system?	What scenarios did user guide explain?	Possible reuse of NFRs, if well documented	Rationalisation of existing requirements, plans	Possible reuse of old glossary etc, if relevant	How measurements were made	(What the priorities were last time)
From standards, templates (Chapter 6)	Which of these roles are relevant?	—	—	—	Which of these NFRs are relevant?	—	Possible adoption of standard terminology, if appropriate	Typical verification approaches for each kind of NFR	—
From trade-offs (Chapter 14)	—	Goals judged not to be worth including	Interfaces judged not to be worth including	Scenarios judged not to be worth including	NFRs judged not to be worth including	Design rationale	—	Adjusting targets to what is attainable in chosen design	Setting of output priorities to what can realistically be attained

and Suzanne and James Robertson's (2004) *Requirements-Led Project Management* [4]. These authors are experienced in advising senior management (see Ralph Young's guest box, 'Project Managers: Direct Additional Focus on Requirements').

Guest Box: Project Managers: Direct Additional Focus on Requirements

by Ralph Young, practitioner and author

The project manager (PM) provides a key role that determines the fate of the project. It's essential that the PM is involved in 'requirements-related work'. Aspects that are essential include:

- **Ensuring that there is an agreed upon vision for the project**, and that its scope has been both determined and limited to something that is 'doable'. Utilize a partnering process from the initial startup of the project to gain and maintain the commitment of all stakeholders to project success. Reduce wasted time and effort during project startup. Ensure that executive sponsorship and senior management support are provided and communicated to project staff. Set expectations concerning the requirements.
- **Coordinating the development of the project plan**, and ensuring that the project plan is used and updated during the execution of the project.
- **Selecting staff**, including a trained, experienced, and capable requirements analyst. PMs must demand and facilitate good requirements analysis, not just tolerate what they get (or worse). Those who are performing requirements work need the support of their PM so that a reasonable proportion of the project's resources (8-14%) is allocated to the requirements process. Utilize a set of requirements-related mechanisms. Recognize and appreciate your staff. Set quality goals.
- **Utilizing effective requirements practices**, particularly, evolving the real requirements before starting other project work, and managing new requirements and changes to requirements. Identify the prioritized, minimum requirements that will meet real needs for the agreed-on vision and scope of the initial delivery. Limit requirements volatility to 0.5 percent per month (6 percent per year), realizing that higher levels will result in loss of control of the project and the inability to deliver it. The PM can help ensure that the right requirements practices, methods, techniques, tools, and templates are deployed for the type of project.
- **Enabling proactive communications for all stakeholders**. Foster and enable effective teamwork. Provide the 'right amount' of discipline and process. Establish an attitude of continuous improvement on the project and ensure that all project staff practices it as second nature in

everything they do. Identify, prioritize, mitigate, and manage project risks. Incorporate a proactive quality assurance role on the project.

- **Minimizing rework.** Work proactively to reduce the average 45% rework on typical projects to 10-15% (by utilizing peer reviews, inspections, and a defect prevention process, for example). PMs must take on the requirements, assumptions and risks that are discovered during the project, and use them to shape and refine the project's plans.
- **Working 'behind the scenes'** to ensure that all stakeholders are identified, and to impress on them how important they are in helping guide the project in the right direction by providing, prioritizing, and checking their own requirements.

Reproduced by permission of Ralph Young.

15.4 Organising the Requirements Specification

15.4.1 Template

Every company, business and industry is different. Many have their own document standards already. You may often find, though, that your document templates actually stop just short of saying anything useful about organising the 'content'.

Once you have discovered the requirements for a project, an appropriate structure will probably suggest itself to you. That will be unique to your project, and far better than any template. Even better, if you sit down with your stakeholders and work out a structure together, you'll build a shared understanding and sense of purpose, which will help to gain buy-in to the project's goals and successful outcome.

The chapter structure of Part I of this book – stakeholders, goals, context, scenarios, NFRs, etc – is a good starting point. That structure is the basis of the requirements template in Appendix D.

15.4.2 Levels

An aspect of organising a specification that is often tricky is getting the level of the requirements right.

For example, stakeholders will often give you requirements at system or subsystem level (making assumptions that may not be warranted about product design), rather than at the level of business need. Such requirements

need to be validated – not blindly accepted – before being assigned to a document. You may also want to find out what ‘real’ stakeholder requirements lie behind the lower-level requirements as stated (see Robin Goldsmith’s guest box in Chapter 1).

Appendix B examines, with examples, the task of assigning requirements to the right level in your documentation.

15.4.3 Can Use Cases Do Everything?

If you are basing your requirements on use cases, watch out for the sometimes strongly held view that use cases cover everything. Other experts argue strongly against depending on one or even a few requirements discovery techniques (for instance, see Table 15.8 on page 392).

For example, systems engineers consider scenarios useful for showing how functions fit together, but not for defining and tracking individual functions. They prefer to trace each function to the scenario steps that involve that function; they point out that identical steps can occur in many scenarios, whereas each function should be defined exactly once. For example, in a retail or financial system, each transaction (whatever its type) must be logged, so a logging step will occur in almost every use case. The many-to-one mapping between scenario steps and functions proves, they argue, that steps and functional requirements are different.

Furthermore, while use cases cover software functions and associated performance requirements well, they may not form a good framework for more or less ‘global’ security and other quality requirements, assumptions, standards, data and other definitions, and constraints (Figure 15.3). Use cases are justly popular in areas where scenarios cover much of the requirements work, as with interactive software applications on a known platform, but that does not mean they are equally effective on other types of project.

You are free to organise your requirements with use cases or otherwise. Either way, you should choose a document structure that matches the natural structure of your requirements, rather than a fixed template.

15.4.4 Organising Product Functions

Several ways of discovering requirements lead directly to an understanding of functionality:

- goals (Chapter 3);
- interfaces and events (Chapter 4);
- scenarios (Chapter 5).

Projects that involve a supplier/customer contract will generally need to translate the discovered requirements into a single contractual list of things

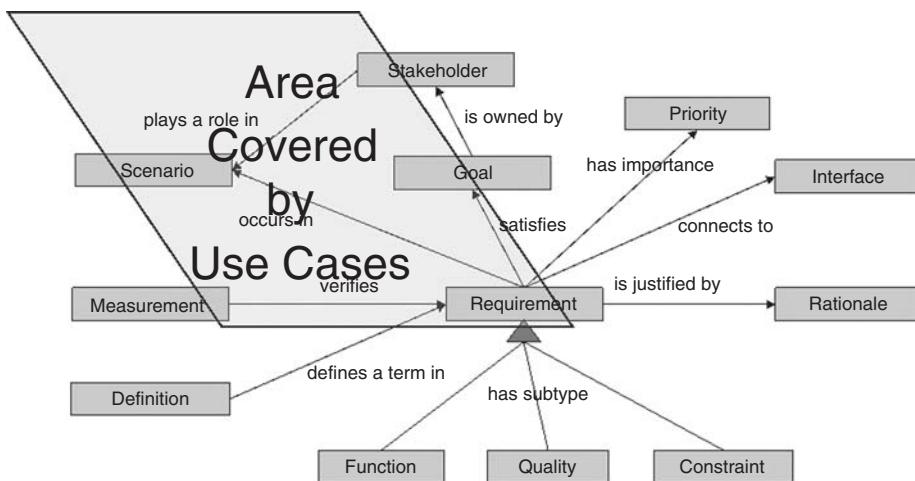


Figure 15.3: Use cases do not cover everything.

someone will have to deliver; in other words, a table of individually deliverable and verifiable items. This list is then checked and enforced.

Projects that do not involve a supply contract (as for in-house development) may find it sufficient just to document their requirement elements (as in Part I of this book), to avoid forgetting anything. The basis for Figure 15.3 is an information model that provides a simple network of these requirement elements.

15.4.5 Traditional 'Shalls'

Answering the question: What individual, measurable, verifiable things must your suppliers/contractors/developers deliver?

The traditional:

'the <abc type of operator> shall be enabled to<do task xyz>'

requirement (described in Chapter 4) is out of fashion in software, at least for functions, but alive and well elsewhere.

Functional 'shalls' are often best organised by scenario, to tell their story as clearly as possible. Scenarios usually form a natural hierarchy of functions. This does not work for specialised scientific software, where there are few scenarios. Instead, there may be an elaborate network of equations and data handling to drive an analysis or simulation, as with working out protein folding, fluid flow or climate predictions.

Other ‘shall’s’ need to be organised, to group related items together, to enable inconsistencies to be detected, and to make the document readable. A hierarchy of document headings helps: for example, you could use the template provided in Chapter 6.

15.4.6 Relating Requirements of Different Types

Requirements may relate to each other at different levels throughout a specification. Consider performance as an example.

Traditionally, performance forms a chapter of its own, among the other usage qualities of a product. In other words, it was treated as an NFR. This generally caused duplication with functions:

- ‘Function 123’ (on page 21 of your specifications):
‘The system shall be able to print address labels.’
- ‘Performance 987’ (on page 543 of your specifications):
‘The system shall be able to print 500 address labels per minute.’

Apart from the effort wasted in duplicating the function in the specifications, this causes further wasted effort in testing. Careful cross-referencing is needed to ensure that the two requirements are updated together if any changes are made.

You can avoid duplication by treating performance as a part or attribute of the function to which it applies, like this:

- ‘Func- 123’ (page 21):
‘The system shall be able to print address labels.’
- ‘Performance’ (same page, next line):
‘500 labels per minute.’

Equivalently, you can use a performance attribute (Table 15.6).

Either way, this approach spreads your performance requirements (or performance attributes of functions) around the functional part of your specifications (e.g. use cases). You then have no separate performance chapter, and no duplication of functions with performance. Designers and testers

Table 15.6: Documenting performance as an attribute of a function.

ID	Function	Performance
Func-123	The system shall be able to print address labels.	500 labels per minute

use the structure of your functional specifications to find your performance requirements – often a convenient route.

However, this happy picture is complicated by the fact that performances may apply at different levels. You might need:

- a performance attribute for a single scenario step or function (like Func-123 above);
- a performance constraint for a whole scenario (or use case);
- a performance target (perhaps a key goal) for a whole product.

At face value, that again means scattering performances all over a specification. Readers have to check not only a scenario step but also its parent scenario and the ancestors of the scenario to see if a performance constraint applied to the step. That is difficult, even with tool support.

We'll Just Put These Elements Into a Template, Then

Well, you could. The elements here are often useful, and they are listed for you as a template in Appendix D. But we hope that the message 'your mileage may vary' is clear.

For example, most projects need measurements (Chapter 9). But if your project's goals are unclear, no amount of measurement will help. You need first to clarify the goals; only then will filling in measurements and rationale and priorities make any sense.

There's also the question of how to organise a template. Some elements, such as stakeholders and goals, are often best documented in separate models (Chapters 2 and 3). Other elements, such as measurements, make sense only when attached to requirements. So, should a template:

- list each element as a separate chapter; or
- treat each requirement as an index card/database record, i.e. a slice across all the elements?

As a starting point, you will do well to focus on your stakeholders, goals, scenarios and so on: these things are helpful on most projects. We've seen projects where scenarios pretty much covered the problem – and other projects where, frankly, scenarios were pretty much useless: the focus was elsewhere.

We urge you to customise the template to suit the needs of your project.

Templates are wonderfully simple, practical things to help you get started on projects. They aren't a substitute for thinking out what your project should do, which is to select what you need and organise it appropriately.

An alternative is a more elaborate structure, such as:

- a family of specification documents (business needs, system requirements, subsystem requirements);
- a goal modelling approach to decompose large (often nonfunctional) goals into measurable functions (see Chapter 3);
- cross-references to show relationships between requirements.

Similar considerations apply to other kinds of requirement, such as qualities. A requirements specification is therefore more like a hypertext with many related elements that you could read in any order, than a book with an obvious start and end.

15.4.7 Conflicting Needs for Requirement Organisation

There is a basic conflict between the needs of stakeholders such as product users (sources of requirements), and development team members such as testers (consumers of requirements):

- Product users need to see that the requirements say what they want to be able to do, so a scenario-based organisation may be best.
- Testers need to see that they have covered each requirement with suitable tests, so separately numbered requirements may be needed to permit traceability.

A scenario – or use case–based organisation with numbered steps is a commonly used compromise. The challenge is that the more complex and ‘instrumented’ you make the use cases to suit developers, the less readable they become, and the less like business requirements.

15.4.8 The Benefit of Requirements (Traceability) Tools

Very likely, given such conflicting needs, there is no single fixed structure (such as a set of use cases) that suits all purposes on a project. Any given document structure is basically one-dimensional: a linear sequence of words in a book. Project information is multidimensional, and may be viewed from different angles by people with different responsibilities.

Consequently, requirements on larger projects generally form a network of elements linked together by traces, in the style of a hypertext or a requirements management tool (see Appendix C).

Good requirements tools can, in addition, provide alternative views of the network of requirements to suit different purposes such as user review, testing and product management (Table 15.7). These go a long way to resolving the conflicting demands placed on requirements.

Table 15.7: Example uses of a requirements database tool by different roles.

To enable this purpose	Display a view containing these items
Review by product operators, marketing, etc	Scenarios beside the goals that they satisfy
Preparation of tests, analysis of test coverage	Tests beside the scenarios that they verify
Product management	Names of scenarios beside product versions that will include them, with status, issues, risks, rationale

15.4.9 An Alternative View: Competing Approaches

We believe, as illustrated in Figure 15.1 and Table 15.5, that requirements should broadly consist of a collection of different elements such as goals and scenarios.

There is a radically alternative opinion: that these elements are essentially competing descriptions of business needs. To put this as sharply as possible: some people believe that discovering, analysing and documenting one kind of element would be enough.

As it happens, schools of thought that favour just one or another of these elements do already exist in both academia and industry. Table 15.8 illustrates some examples of these schools of thought, along with some of the weaknesses of each single-element approach.

Table 15.8: Competing approaches for discovering business needs.

Elements used in approach	How this approach describes need	Schools of thought that favour this approach	Disadvantages of using this approach alone
Stakeholder analysis (Chapter 2)	Identifies the political, economic, social, and cultural forces that drive design	<i>Soft systems methodology</i> (informal models such as rich pictures)	No precise, verifiable description of system requirements; unsuitable for contracts
Goal modelling (Chapter 3)	Says what stakeholders want	<i>KAOS</i> project (logical goal decomposition); <i>i*</i> (goal modelling language)	Hard to express timing relationships (scenarios) between goals; danger of diving into design without considering alternatives
Event-driven analysis (Chapter 4)	Identifies events at interfaces, and says how to handle them	<i>Event-driven methods</i> (see book list in Chapter 4)	Assumes context is well-defined: does not attend to soft systems issues or conflicting stakeholder goals

Table 15.8: (Continued).

Elements used in approach	How this approach describes need	Schools of thought that favour this approach	Disadvantages of using this approach alone
Scenario analysis (Chapter 5)	Says how design will deliver results to human operators	<i>Cockburn-style use cases; agile</i> (user stories)	Over-emphasis on product behaviour; risk of ignoring qualities and constraints from nonoperational stakeholders; danger of diving into interaction and user interface design without justification or clear priorities
Reuse – standards and templates (Chapters 6, 13)	Defines generally needed qualities, constraints, and procedures for achieving them	Standardisation, regulation, quality assurance	Functions and innovative aspects specific to the new product are not covered
Rationale modelling (Chapter 7)	Explains why design is needed, or what is needed to make it safe	<i>Compendium</i> project and tool (informal rationale diagrams); <i>GSN, CAE</i> (safety case diagrams)	As for goal modelling, lacks timing (scenarios); danger of deciding on a solution and devising reasons that support it
Data modelling (Chapter 8)	Defines the data, rules and relationships to ensure business works properly	<i>UML class modelling; traditional entity-relationship data modelling</i>	Lack of vision of end-to-end behaviour (scenarios), context, purpose
Measurement (Chapter 9)	Shows exactly what results the design must provide	Traditional list of 'The system shall ...' (requirements)	Requirement wording has to carry whole burden of defining terms, context, scenarios, timing; easy to lose sight of stakeholders, goals and purpose
Priorities, trade-offs Chapters 10, 14)	Shows which design features are needed most, or would be most profitable	<i>Business case; benefit/cost analysis; value engineering</i> (spreadsheets)	Suggests requirements are independent and can be compared financially; tends to ignore context, purpose, qualities, constraints, nonfinancial beneficiaries

We suggest that a more balanced mix of requirement elements is likely to be better for most projects. Overlapping approaches give you more opportunities to discover missed requirements, and provide a more natural framework for expressing different aspects of a situation. Table 15.1 suggests that the mix varies for different types of project. Table 15.9 summarises our (inevitably general) suggestions for tools, templates and techniques for documenting the requirement elements. We reiterate that your project may not need to use them all, but it will very likely benefit from using several of them.

We believe that there is now enough evidence from experience and research to show that different combinations of requirement elements – and effective ways of discovering them – suit diverse contexts.

A specification is a system with a special emergent property: communicating a coherent and informed understanding of the requirements. That property emerges when the requirement elements for a project are assembled and interlocked appropriately.

Table 15.9: Tools, templates and techniques for documenting requirement elements.

Elements	Techniques for documenting the element	Tools and templates ('RM' = requirements management)
Stakeholders (Chapter 2)	Stakeholder onion models	Alexander's template (Chapter 2); Diagramming tool
Goals (Chapter 3)	List of goals; Goal models	Spreadsheet or RM tool; Diagramming tool; i* (goal modelling language)
Context, interfaces, scope (Chapter 4)	Checkland-style rich pictures; Context diagrams; List of events	Diagramming tool; Spreadsheet or RM tool
Scenarios (Chapter 5)	Storyboards Operational scenarios; Use cases; User stories	Word processor or RM tool; Cockburn's template (Section 5.3.4, Use Cases in Chapter 5)
Qualities and constraints (Chapter 6)	Hierarchical list of qualities and constraints; Goal models	Word processor or RM tool; Alexander's checklist (Chapter 6); Lauesen's template (Chapter 6, Guest Box); Standards, e.g. ISO/IEC 9126, IEEE 830

Table 15.9: (Continued).

Elements	Techniques for documenting the element	Tools and templates ('RM' = requirements management)
Rationale and assumptions (Chapter 7)	Justification attribute of requirements; List of assumptions; Traces to goals, assumptions, etc; Rationale diagrams; Safety case diagrams, e.g. GSN, CAE	RM (traceability) tool; Compendium (rationale modelling tool); GSN, CAE tools
Definitions (Chapter 8)	List of acronyms; Project dictionary with traces to requirements, etc; Roles and responsibilities table; UML class diagrams or entity-relationship models (domain/data models)	Spreadsheet, word processor or RM tool; UML/other diagramming tool
Measurements (Chapter 9)	List of 'The system shall ...' requirements	Standards, e.g. IEEE 830
Priorities (Chapter 10)	Priority attributes of requirements	Spreadsheet or RM tool; Boston Matrix of benefits and costs

15.5 The Bare Minimum of Putting it all Together

- Choose the simplest process that works for you.
- Scan through each element described in Part I of this book. Discard any that are not relevant to your project. Do the minimum of each element that remains.
- Then do just enough prioritisation (Chapter 10) and trading-off (Chapter 14) to ensure you can realistically build what you promise.

15.6 Further Reading

15.6.1 Choosing and Tailoring Development Life Cycles

1. Stevens, R., Brook, P., Jackson, K. and Arnold, S. (1998) *Systems Engineering, Coping with Complexity*, London: Prentice Hall.

Stevens et al is a good place to start for a general understanding of the development life cycle.

2. Farncombe, A. (2004) Project Stories: Combining Life-Cycle Process Models. In Alexander, I. and Maiden, N. *Scenarios, Stories, Use Cases*, Chichester: John Wiley & Sons, Ltd.

If the basic Stevens-style life cycle does not seem right for your project, read Farncombe's discussion of how to tailor-make your own life cycle by combining different life cycles. The basic patterns discussed are: evolutionary, to permit rapid change from one product to the next; incremental, to permit rapid delivery of at least some functionality when the overall architecture is fixed; and high-risk, which permits development in a context which would be too risky any other way. Farncombe discusses how far these can be combined to suit your project's situation.

15.6.2 Managing Projects From Requirements

Here are some books to take you further once you have discovered the requirements for your project. They are less about 'requirements management' in the sense of organising the requirements in a database, than about managing your project to implement the requirements successfully.

3. Young, R.R. (2001) *Effective Requirements Practices*, Boston: Addison-Wesley.

Ralph Young's book is about everything you need to do on your project to make it work. It's about the processes you choose to discover your requirements and to verify them; it's about handling change; it's about iterating 'repeatedly'; it's about measuring how well you are doing; and it's about continual improvement. It is practical, down to earth, and aimed at managers of large and complex projects.

4. Robertson, S. and Robertson, J. (2004) *Requirements-Led Project Management: Discovering David's Slingshot*, Boston: Addison-Wesley.

The Robertsons' book, too, is aimed at project managers, but it is a far more conversational book than Young's. In a way, it is a follow-on book to their popular *Mastering the Requirements Process* (2006), but that book was essentially for requirements analysts, and dealt with managing the requirements themselves. This book references the various requirements activities, but from the point of view of working out the right process for your project as a whole.

Both Young and the Robertsons would agree that managers need to understand requirements processes.

15.6.3 Classics for Inspiration and Reflection

We began this book by saying that requirements are simple but not easy. Perhaps, after several hundred pages, it no longer seems so simple. A good place to end may be with some classic books that take a timeless look at finding out what is wanted, and delivering it.

5. Vincenti, W.G. (1990) *What Engineers Know and How They Know It: Analytical Studies from Aeronautical History*, Baltimore and London: Johns Hopkins.

Vincenti's fine book is a history, but a gripping one because it shows how, in the concrete example of an evolving modern industry, knowledge actually grew. The industry did not follow a neat, smoothly programmed path, but lurched in fits and starts from a quite remarkable degree of vagueness to an equally impressive solidity of practical knowledge. Also fascinating is the careful distinction between scientific knowledge, gathered by controlled experiments, and engineering knowledge, developed with practice and experience of what works.

6. Dreyfuss, H. (1955) *Designing for People*, New York: Simon and Schuster. Henry Dreyfuss' informative and beautifully illustrated book shows just how much our understanding of design and human-machine interface requirements has changed in half a century. The book is elegantly laid out, and printed and bound with an attention to detail that suggests a less hurried age. The breadth of skills and roles that one 'designer' could take on in those days is almost breathtaking. What has not changed at all is the need to bridge the gap between the world outside and the world of engineering.

15.6.4 A Look Ahead

7. Levine, R., Locke, C., Searls, D. and Weinberger, D. (2000) *The Cluetrain Manifesto, The End of Business as Usual*, London: FT.com.

Finally, having taken a look back into the history of technology, we should perhaps take a look ahead. *The Cluetrain Manifesto* is about marketing – a profession that certainly doesn't consider itself anything to do with 'requirements' or 'engineering', yet which is all about understanding the real requirements for current and future products. Levine and his colleagues offer fresh insights into the limitations of conventional marketing practices. Maybe we can learn from them.

APPENDIX A

Exercise Answers and Hints

The answers and hints here are meant to give you some reassurance that your answers to the exercises in the chapters are on the right lines, and to suggest useful directions for further study.

Chapter 2

- a. **Key stakeholders:** Directors and shareholders in your company (financial beneficiaries); your company's sales and marketing (sources of requirements); customers (will be functional beneficiaries and operators of the cutting machine, making them industrial users – a hybrid role somewhat different from mass-market users); the government regulator of industrial health and safety in each country you plan to sell the machine; your competitors (negative stakeholders). If your company intends to maintain the machines, then your maintenance team will be major stakeholders. Your company's spares and distribution management is also important.
- b. **Analysing and validating the stakeholder list:** Check and update the list regularly; review and follow through actions, e.g. meeting key stakeholders.

Chapter 3

- 1a. **Major goals:** Bring machine to market on time and to budget; develop new machine; obtain safety certificates in all target countries; understand requirements in each target country; prepare user and sales documents in all target languages; ensure sales department is prepared; ensure maintenance department is prepared; ensure spares and distribution department is prepared; understand competitors; develop competitive strategy.
- 1b. **Likely sources of tension:** Development time versus the need to beat competition; sales agreements (features sold to customers) versus design trade-offs (what can be built in the time); time-to-market versus completeness of user documentation in all languages; international sales versus time to set up spares distribution and maintenance in all target countries.

Chapter 4

- a. **Context and b. Events:** Your answers will depend on the business model you choose.

If it has an elegant ambience with high quality food prepared by a chef, then you will buy raw ingredients from many different suppliers, perhaps straight from farmers, and you will manage bookings (business events) months in advance from corporate and personal customers.

If it's a fast-food model, then you may have just one wholesale supplier, or perhaps you have a franchise from a well-known brand, and you won't take bookings at all.

Chapter 5

1. **Scenario:** There are many possible scenarios. For the climber's altimeter, the main goal is to display altitude continuously. If you choose a GPS altimeter, it will only be accurate to about 10 metres, which might not be enough if you mean to guide a climber through a tricky route. Your scenarios may allow the climber to scroll forwards or back from where the device thinks it is, to find the next climbing move.
2. **Exception:** That facility will also be useful if the GPS signal is lost, an important exception that must be handled carefully. You might think of providing a 'dead reckoning' feature to allow the climber to enter an estimate, e.g. she reckons she has climbed 20 metres from a known point, for instance. If you choose a barometric altimeter, it will be accurate enough to provide detailed vertical guidance, though GPS will still be needed for horizontal position (i.e. the device locates position on the climbing route and indicates the next climbing move), but only if calibrated.

Guarantees: If the altimeter is working, the success guarantee is to display altitude to within a stated limit (or perhaps, to display the current uncertainty, e.g. with an error bar). If the altimeter is not working, the minimal guarantee is to avoid displaying a misleading altitude (e.g. to show time since it last made a reading).

Chapter 6

Suppose your target is **usability**.

1. **Goals:** Your goals for the restaurant could include making the waiter's ordering system usable with minimal training. The key point is that the large-scale goal needs to be broken down into different areas of the business for it to be of practical use.
2. **Acceptance criteria:** You could measure usability by specifying e.g. that 90% of new waiters can place a correct order to the kitchen after one hour of training. You could also consider requiring the system to have a computer-based training/testing mode built in, to bring new waiters up to an acceptable level of skill with the ordering system.

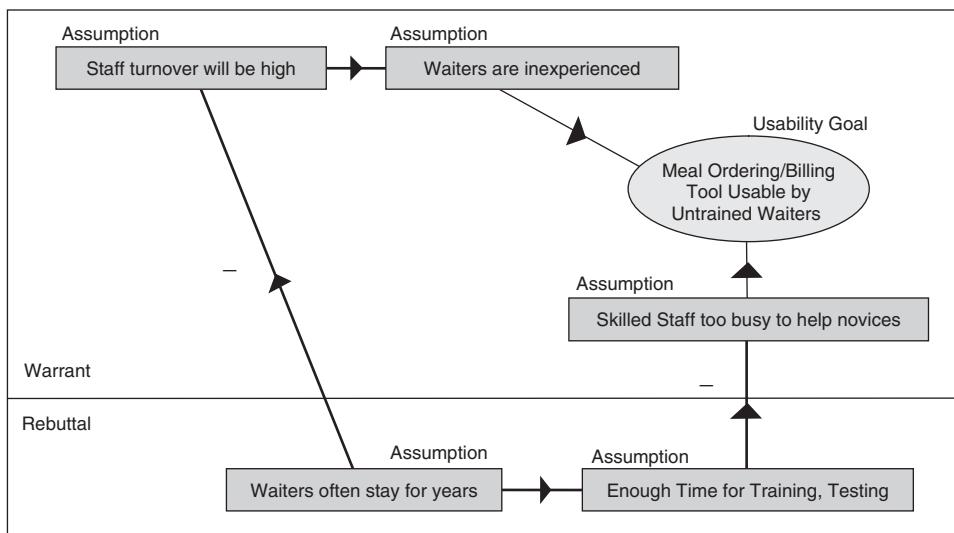
Chapter 7

Rationale model: Suppose you are arguing the case for demanding high usability of the waiter's ordering system by untrained staff (see the exercises for Chapter 6).

The warrant: Staff turnover may be high. Waiters may have no previous experience (e.g. students paying their way). Experienced staff may be too busy to help novices.

The rebuttal: Waiters often stay for long periods. Training and testing can be provided. Novices can ask experienced staff for help.

A possible answer:



Chapter 8

Definitions: Look through the definitions you have written, and check that all terms you have mentioned in definitions are highlighted (with italics, capital letters etc, according to your convention). Check that all such terms are themselves defined or designated. Check that the definitions make sense in terms of your other definitions.

Designations: You should have written designations for a small number of things in the world that your definitions relate to. For the restaurant's IT system, such things include the restaurant itself, and the tax and public health authorities.

Chapter 9

Deciding on measurement values: All the values involve trade-offs between the cost and feasibility of implementing them, against the value of the requirements to their outdoor leisure customers.

- Weight** (product mass) is a critical parameter for any portable device. A limit on weight forms a budget constraining almost every function, so it powerfully affects cost as well as functionality. A heavy device will be hard

to sell – as will a device that does not do much. A suitable upper limit can be found by: market analysis; trials with a prototype; looking at rival and analogous products. The final trade-off will involve the whole development team and a decision by product and business management.

- b. **Temperature** limits can be found from scenarios for the product (e.g. ice climbing) combined with meteorological data. The trade-off is with the operating limits of the technologies to be used in the device, e.g. CMOS electronics.
- c. **Waterproofing** quality affects cost. If none of the scenarios actually involves diving, then the requirement is essentially to survive shallow immersion (e.g. in a canoe), possibly for hours at a time. Inadequate waterproofing is a likely cause of product failure, so three atmospheres (about 30 metres depth of water) is a wisely robust choice, if it can be afforded.
- d. **Shockproofing** similarly affects cost. Again, rough handling is a likely cause of failure, and robustness an important measure of product quality for an outdoor leisure device, so one metre might not be enough. To identify the right value, see ‘a’.
- e. **Recharge** time is a practical matter but perhaps not the most critical issue. If recharge is by winding and the device is for amateur use, 30 minutes is a very long time. Conversely, if there is a solar cell or if the device is for expedition use, 30 minutes might be conveniently short. For comparison, the Summit Freeplay radio plays for 30 minutes after a 30-second wind: perhaps full recharging is not the essential issue here (it doesn’t matter if the batteries are only partly charged). Again, to identify the right value, see ‘a’.

Chapter 10

Listing input priorities: Each scenario you discovered in the Chapter 5 exercise should have a priority. Mountain walkers and climbers will probably agree that an altimeter is desirable; canoeists and others will probably not see it as a benefit personally, and may want it to have a lower priority.

Chapter 11

1. **Observing a colleague:** Whatever you observed is of interest. The value of this exercise is in reflecting on its lessons for requirements discovery. What did your colleague do that did not exactly fit the standard scenario for the task? What else happened – were they interrupted by another task? If you just used a notebook and pen, did you realise you needed something else, like a voice recorder or a camera? Was your colleague able to work normally despite your observation?
2. **Interviewing multifunctional device stakeholders:** This exercise is interesting, both for what you might learn from the interviewees, and for what you find out about your own technique. Did you hear any requirements

you would never have thought of yourself? Did the interviewees express themselves easily? How did you introduce the interview? Did you use something to 'break the ice'? Did you tell the interviewees the right amount about the 'project'? What would you do differently next time?

You might take along a prototype MFD if you have one, or else an analogous product, such as a multifunction digital camera or mobile phone; or perhaps one or two traditional outdoor leisure tools such as a (multifunction) Swiss Army knife, a magnetic compass (which incorporates a ruler, a map scale, and a magnifying glass), a whistle, etc.

Chapter 12

1. **Groupware for multi-national project:** Since telephone conferencing is not working, you clearly need to go further. Just adding video may not help very much – it certainly won't make scheduling the conference times any easier. You need to look into tools (take a look on the Web!) that enable models and data to be shared, debated and commented by all the teams. For instance, the interface definitions need to be visible for commenting and, where appropriate, editing so that direct technical progress can be made more smoothly. Get the choice of tools approved by California, who seem to have problems understanding the existing proposals (so getting California engaged in the choice of tools is crucial).
2. **Recommended measures:** It sounds as if getting some team members co-located, probably in California, is going to be necessary. A programme of visits to the other project sites is another strong option. Hold a workshop in India or Europe, with two or three members from each team focusing on context and scope.

Chapter 13

1. **Gathering ideas from competitors:** Maybe there is no exact rival (happily). In that case, look for elements of unrelated products that may be of interest. The price, weight and battery life of such products may be key. Find reviews (e.g. in magazines) of those products, to see what they lack in the eye of the users.
2. **Making a first prototype:** What does your prototype need to show? The user interface of the multifunctional device may be the whole of its surface, just as mobile phones nowadays have buttons, lamps, camera lenses and interfaces on all six surfaces. Perhaps you should make a (large) box, labelled with what goes where.
3. **Showing the prototype:** Remember that people comment very rapidly on a prototype, and you get just one chance to see their initial reaction. Perhaps you could film them (a mobile phone video will do fine), or have a colleague make notes while you demonstrate the prototype.

4. **Making a better prototype:** If people give you direct suggestions, you need only work out how to implement them in your prototype. If they offer a criticism like, ‘But that’s awkward to reach’, then you need to act out the same scenario and explore what works better.

Chapter 14

- a. **Trade-off matrix:** The exact scores are subjective, but it will probably look something like this:

Criterion Option	Surroundings	Food	Service	Cost	Risk
Young French	+++	+++	++/+++	+	++
Traditional hotel	++	+	+++	+	+++
Personal catering	+	+++	+	+	++
Self + hired waiters	+	+	--	+++	---

- b. **Show-stoppers:** The ‘self+hired waiters’ option is dangerously risky and can be triaged out.
- c. **Rationale:** The personal catering option cannot compete in terms of surroundings, but offers good value and excellent food at low risk. The traditional hotel option has excellent service and nice surroundings but fails to meet the requirement for original food. The young French option is the best all-rounder, and is the only one to offer fine food, elegant surroundings and most probably fine service all together, so it is the winner.

Appendix B

- a. **Vibrant and fast moving:** This is a publicity statement. It has no place in the requirements.
- b. **Compatible with user interfaces:** this is trying to be a Level 3 product quality, but it is not verifiable, even without the ‘remorseful’ afterthought ‘and similar games consoles’.

Indeed, since the user interfaces of the named platforms are different (the Wii has a controller that is waved in the hand, while the Xbox 360 has a controller that stands on a table), implementation would presumably require a different product version for each platform.

Perhaps there is a Level 2 stakeholder goal here, but more likely it is a Level 0 business objective (to take some market share from the famous games consoles? But then compatibility makes no sense) masquerading as a requirement. It sounds as if further discovery work is needed.

- c. **Beginner and advanced levels:** This is a Level 3a product feature, which can be verified by test or demonstration. Whether the implemented levels are really suitable for these ages can be determined by usability testing, or

possibly by an experienced user panel. The panel may usefully be able to predict suitability from storyboards (scenario prototypes).

- d. **Retain new customers:** This is a Level 1b project management target, as it measures what the project vision is to achieve.

APPENDIX B

Getting the Level Right

By Joy Beatty, Seilevel, and Ian Alexander

Reproduced by permission of Joy Beatty, Seilevel Inc.

A constant challenge for projects is to get their requirements at the right level for the purpose at hand. There is a tendency to jump prematurely into design when product requirements are wanted; and to write product requirements when business goals are sought.

The body of this book in general assumes you are trying to discover requirements at stakeholder level. But during discovery, you will certainly find requirements at a mixture of levels, and will need to sort them out.

This appendix suggests a typical set of levels, explores some examples and sets some exercises for you to practise on. We hope it will help you get your requirements at the right level for each phase of your project.

Requirement Levels

There is no academic or industry-wide agreement on what the different levels at which requirements can be expressed are, what these levels are called, or even how many of them exist.

Traditional Levels in a Customer-Supplier Setting

The traditional levels for requirements where a single customer commissions the development of a system from a single supplier are:

- **User requirements**, also known as ‘business requirements’ or ‘stakeholder requirements’. These are meant to describe the business need for the product or system under design from the point of view of people who want results from it. They may be written by a customer organisation as part of its invitation to tender, or as part of its contract with a supplier.
- **System (or software) requirements**, also known as ‘product requirements’ or ‘functional requirements’ (even though they always contain nonfunctional elements too). These are meant to specify the product or system under design, given that a design approach has been chosen. They

may be written by a supplier as part of its proposal to, or contract with, a customer.

These traditional levels work well in a customer-supplier setting. Additional levels are needed when suppliers in turn need to subcontract out parts of their work, forming a 'supply chain'. Levels below system may be called 'subsystem requirements', 'component requirements' and so on, each one forming a part of a contract.

Requirement Levels for a Mass-Market Product Company

If you are developing a set of mass-market products, you do not have a single customer to give you 'user requirements'. You have to act as internal customer, on behalf of the mass market. The following set of levels may help you to organise your requirements. The relationship of these levels to the traditional ones is indicated where possible.

Note that you can expect to have very few items at the higher levels (0 .. 2 especially).

Level 0: Business Objectives (across several projects)

Alternative names: sales targets; senior management goals; business targets.

These are *above* the level of any single project. They describe metrics the business must meet in order to solve one or more problems. They may be met by a combination of business activities such as sales and marketing, and one or more development projects.

E.g.: 'Increase number of customers by 10%.' (via several projects)

Level 1a: Project Vision (just one of the projects)

Alternative name: mission statement.

This is the top-level statement of what the project is, where the project is a proposed solution to accomplish one or more of the business objectives. It is the root goal in a project's goal model.

E.g.: 'Develop a fun-for-the-family game for the Zap! Console.'

Level 1b: Project Management Targets

Alternative name: product targets.

These are measures of what the project named in Level 1 is to achieve.

If this is the only project, then these targets directly measure the Level 0 business objectives. If this is one of several projects, then this level shows how the projects are each to contribute to achieving what is required at Level 0.

E.g.: 'Game is ready to be sold by start of next December.'
'Children under 10 rate the game's fun score an average of seven or higher.'

Level 2: Stakeholder Goals

Alternative names: goals; objectives; roughly equivalent to 'user requirements'.

These may be direct statements by beneficiaries of what they want; or by surrogate stakeholders, such as product management and marketing, of what they believe the market wants. These are useful guiding principles for a project, but if you mean to use them as requirements in a contract, you must add acceptance criteria (see Chapter 9, Measurements) to make them verifiable.

E.g.: 'Five-year-olds understand game rules.'
'Game is free from violence.'

Level 3a: Product Features

Alternative names: product functions; equivalent to part of 'system requirements'.

These are functional design elements that, through prioritisation and trade-off, have been included in a given version of the product.

E.g.: 'Simulated rolling of dice.'
'Onscreen hints can be turned on and off.'
'Film-quality mood music.'

Level 3b: Product Qualities

Alternative names: NFRs; '-ilities'; equivalent to part of 'system requirements'.

These are desired qualities that, through prioritisation, have been included in a given version of the product.

E.g.: 'Portable to Apple Mac operating system.'

Level 3c: Design Constraints

Alternative names: equivalent to part of 'system requirements'. May include interfaces.

These are specifications that insist on specific implementation choices. We would caution, though, that one man's specific design is another man's vague requirement.

E.g.: 'System construction shall be as in Figure 4.'
'Permanent storage shall be provided by a Foxton Fastrak 3.5"

SATA-300 7200 rpm 80 GB hard disk drive.'

'User interface screens shall be as shown in Annex 3.'

Using Levels to Help Discover Requirements

Understanding and applying these levels serves a purpose in discovering the requirements (Figure B.1).

Establishing Level 0 is critical for the success of a project, because it is used to define and control its scope (typically, alongside other 'sister' projects).

Levels 1 and 2 represent steps in a thought process that is helpful to make the bridge to Level 3. They are not necessary on all projects, though. If your team can confidently get to product features and qualities without defining a project vision, then you should skip to Level 2 or 3.

Examples

Too High

Attempt: 'The game shall be easy for family members to play.'

Comment: This is written as a product quality, but it is not verifiable as it stands.

Actual Level: Level 1 or 2: an unquantified goal with no definite stakeholder as owner.

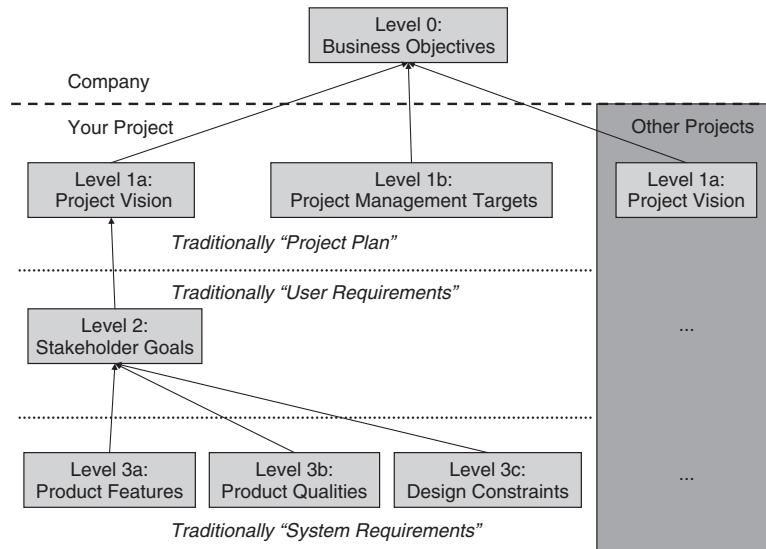


Figure B.1: Requirement levels.

Suggestions: Usability testing is essential here. If the product is intended for children aged between five and nine, then properly supervised testing with families with children of that age is required.

Redraft: ‘The game shall be playable by families with children aged five to nine, with the family able to begin play within three minutes of starting the game for the first time. Acceptance criteria: passes usability test.’

Too Low

Attempt: ‘Game delivers additional user satisfaction through use of a service-oriented architecture (SOA) for all software components.’

Comment: This is framed to resemble a stakeholder goal but is actually a design decision. The first part is an unrealistic rationalisation to make the second part (SOA) seem more like a ‘user requirement’.

Actual Level: Seemingly Level 3, but SOA as a feature is not visible to players; the stakeholders who might want it are system architects and perhaps software maintainers.

Suggestions: This should probably simply be deleted from the requirements.

Redraft: Explain in the product’s design rationale why SOA meets the requirements better than other architectures (ideally, by documenting the trade-offs – see Chapter 14, Trade-offs).

Exercise

Identify the level of the following draft requirements:

- a. Vibrant and fast moving platform that surpasses customers’ expectations by consistent focus on design, creativity and the business model.
- b. Compatible with user interface of Nintendo Wii, Xbox 360 and similar games consoles.
- c. Game provides beginner (age 5–6) and advanced (age 7–9) levels.
- d. Retained 80% of newly acquired customer base, one year later.

Hint: the answers could include ‘too high’ or ‘too low’, as well as a numbered level.

APPENDIX C

Tools for Requirements Discovery

Answering the questions:

- How far can tools help in discovery?
- What tools should we use to help discover our requirements?

This appendix looks, in turn, at:

- graphics tools for requirements modelling;
- simple tools for creating requirements;
- requirements management tools.

Graphics Tools for Requirements Modelling

Any kind of diagram that acts as a model can be supported by a suitable graphics tool.

Specialised tools are generally not essential – you can use a simple drawing tool instead, unless you need to trace to individual elements in a diagram/model, or you have other specific integration needs in your project environment.

No Magic Bullet

Tools are no use if they are worn around the neck as amulets. Having the best requirements tool does not of itself solve any part of the requirements problem, let alone requirements discovery.

Too Tidy to Be True?

There is danger in the use of tools (not necessarily with tools themselves). Tools create very neat, tidy diagrams. These can give a misleading impression of correctness and completeness, of having been cooked up by The Men Who Know. The audience may be cowed into submission by the dazzling technical accomplishment (and thus may not state their points of view).

Or, they may object on the grounds that they were not involved, and that they see things differently. Tools risk interfering with a critical element in the requirements creation process: the messy, participative informality that contributes to agreement and team understanding.

Deliberately Untidy Tool Use

Perhaps there is value in creating deliberately rough models even if you are using tools. Show some empty boxes, arrows that don't go anywhere, etc – anything to make it clear that the model isn't finished!

Informality is not always possible. If you have a large team, a long timescale (so that people join and leave), geographic spread (e.g. outsourcing, subcontractors) or the need for public consultation, then you just can't involve everybody informally. In such cases, you will have to make everything neat and tidy, in cold print and slightly warmer graphics. But all that comes after requirements creation, inevitably by a relatively small core team. There is a real risk of losing participation, as people on the outer fringes feel that their views are not being heard.

A good rule of thumb is that scruffy is best during requirements creation. Chapter 12 (Requirements from Things) discussed the need for quick-and-dirty prototyping, for instance. Corporate graphics, neatly formatted text, artists' impressions, engineers' blueprints and analysts' UML models can come later.

Simple Tools for Creating Requirements

Without Computers

The best tools for creating requirements (and this is supported by many people's experience in user interface and other kinds of design too) are the simple ones:

- pens;
- paper;
- flipcharts;
- whiteboards.

Sticky notes, non-UML stickmen, clipart and photographs can help, too.

Anything that helps people to grasp the essence of the problem and avoid getting lost in details ('going down rabbit-holes') is just fine. Modelling tools can perform dazzling tricks, but they are splendid ways of avoiding talking to stakeholders, or busily rearranging the deckchairs on the Titanic as the ship heads blindly into the ice-field.

When you're trying to work out where your project should be going, it's best to keep the tools simple and human, and your eyes open for icebergs. If somebody says something that doesn't fit your ideas, you can just note the fact and work out what to do later.

How Not to Use Tools in a Workshop

There is nothing more off-putting than sitting in an audience while somebody with a computer sits above you on a stage, madly trying to fit your ideas into their inadequate conceptual framework, which is boldly projected onto a screen for all to see.

Notes can be recorded just fine on paper, in a way that supports participation. Structuring ideas takes time, and is not best done while 100 people are getting bored and critical.

When Software Helps

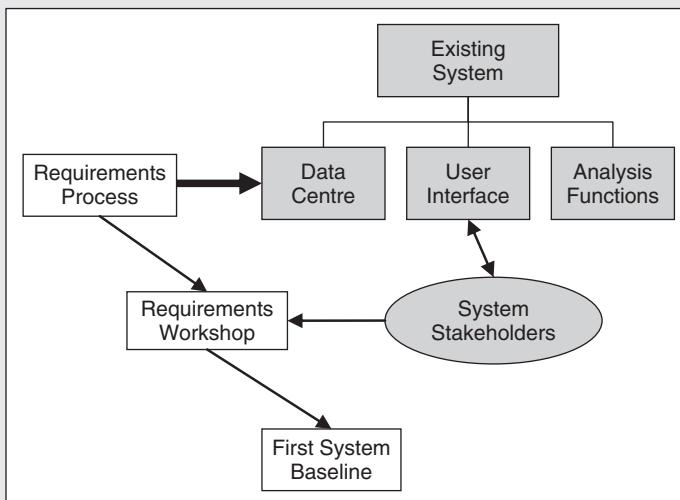
Afterwards, alone at your desk, it is well worth creating models of different kinds, and formalising these using software tools. This is because:

- models (especially if diagrammatic) give you, the modeller, immediate visual feedback on completeness and balance (have you covered some areas but not others?);
- models, visual or not, allow you to check in various ways for correctness:
 - e.g. a systematic project dictionary is a nondiagrammatic model of the domain, actors and products. It allows you to check that terms have all been defined, and are used consistently;
- explaining a model to stakeholders is a useful way of validating your requirements;
- models with clear semantics, even if quite simple, help you think clearly and systematically about a problem;
- presenting clear and accurate models after a workshop confirms that you have listened attentively to what the stakeholders said;

- some people (but not all) find diagrams much easier and quicker to read than the equivalent (inevitably rather dry and monotonous) list of statements in text form.

The Danger of Loose Models

Loose models, e.g. those consisting of boxes and arrows with no particular meaning, are dangerous in requirements creation. Here is an example:



Do you feel you know the semantics of this diagram?

If a box sometimes means an activity or business process, sometimes a department, sometimes a stakeholder and sometimes a piece of data or a product, while the arrows variously mean communication, a development process, and/or the production of data, then the diagram may well give people a feeling of (nearly) understanding, while actually it conceals a mass of problems.

Many kinds of models can be used for discovering requirements (Table C.1). Several have been mentioned in the course of this book. Others include software analysis and business models, but can be equally useful: do not fall into the 'not invented here' trap; things may be valuable even if your organisation or profession does not generally use them. Anything that helps is a good thing.

Table C.1: Many types of model can be useful for requirements discovery.

- **Requirements discovery models:**

- stakeholder onion models (Chapter 2)
- stakeholder influence diagrams (Chapter 2)
- goal models (Chapter 3)
- use cases (Chapter 5)
- Checkland-style rich pictures (Chapter 4)
- rationale models (Chapter 7)

- **Software analysis models:**

- object models (of parts of a domain) (Chapter 8)
- state transition diagrams

- **Business models:**

- flowcharts (UML activity diagrams)
- decision trees
- organisation charts (organograms)
- risk models

UML in Requirements Work

The standard set of UML diagram types is not very well suited to requirements work. UML is primarily intended for software specification – now gradually being broadened out to include systems of other kinds – once the requirements are known. Many of its special features, e.g. for detailed analysis of timing and message passing, are of little use for requirements discovery.

Conversely, many aspects of requirements creation, like stakeholder analysis, goal modelling, rationale analysis, prioritisation and creating acceptance criteria, are not conveniently handled with UML diagrams.

In requirements work, there is no need to be at all ideological about diagram types; some famous analysts make effective use of older types of model, such as traditional context and dataflow diagrams.

Simple Computer Tools

You probably already have several software tools on your computer, able to help you create useful models of your requirements.

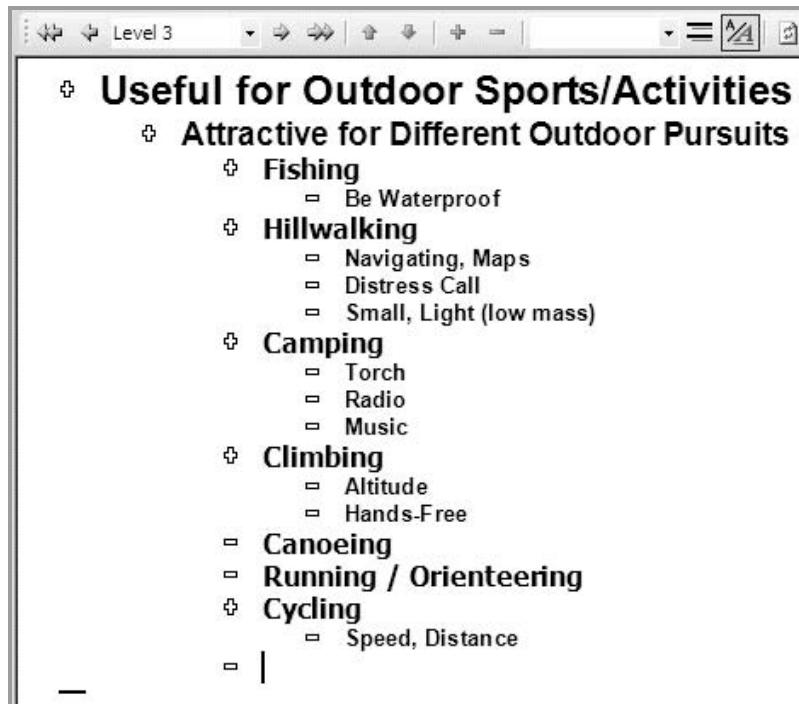


Figure C.1: A goal hierarchy under construction with the Outline tool.

Microsoft Word's Outline tool (choose **Outline** from the **View** menu) allows you to edit a hierarchy of headings quickly (Figure C.1). You can move items up, down, left and right (i.e. promoting and demoting them in the hierarchy) with the buttons shown at the top. This may be sufficient to organise lists of goals, stakeholders, assumptions, NFRs (qualities and constraints), etc. Considering that nearly everybody has Word on their computer, it is amazing how rarely Outline is used. In effect, you already have a basic mind-mapping tool on your desktop.

The example in Figure C.1 does also show a limitation of hierarchy tools. If you wanted to add 'Be Waterproof' as a subgoal under 'Canoeing', you'd have a duplicate of the one under 'Fishing'. Thus, goals form not a hierarchy but a network, so a hierarchy tool is just a starting point.

Microsoft PowerPoint allows you to draw diagrams using predefined 'autoshapes', which include both 'basic shapes' such as boxes and ellipses, and connectors (lines, straight or curved, with or without arrowheads) that automatically stick onto the basic shapes. The effect is to make it easy to drag shapes into position, to resize and label them, so as to make requirements models such as goal and rationale diagrams without too much effort. Flowchart symbols are also included. Obviously, any such models are stand-alone diagrams; there

The screenshot shows the DOORS software interface. The main window displays a table titled 'Goals' with columns for 'ID', 'Example Goals in DOORS', 'Goal Type', and 'Goal Obstacle Type'. The table lists various goals, each with a hierarchical tree icon to its left. A mouse cursor is hovering over the row for 'G-17 2.15 Affordable Price'. The 'Goal Type' column for this row shows 'Commercial' and 'Goal'. The 'Goal Obstacle Type' column shows 'Obstacle'. The bottom status bar indicates 'Username: Ian' and 'Exclusive edit mode'.

ID	Example Goals in DOORS	Goal Type	Goal Obstacle Type
G-2	2 Spacecraft Goal Model Use Goal Model Editor to create diagrams.	Functional	Goal
G-3	2.1 Do Worthwhile Space Science	Functional	Goal
G-4	2.2 Launch into Space	Functional	Goal
G-5	2.3 New Instrument Designs	Functional	Goal
G-6	2.4 Long Lifetime	Reliability	Goal
G-7	2.5 Low Mass	Functional	Key Goal
G-8	2.6 Very Reliable	Reliability	Key Goal
G-9	2.7 Keep on Station a Long Time	Functional	Goal
G-10	2.8 Low Power Consumption	Functional	Goal
G-11	2.9 Cautious Design	Functional	Goal
G-12	2.10 Large Supply of Thruster Fuel	Functional	Goal
G-13	2.11 Many Different Instruments	Functional	Goal
G-14	2.12 Long Time for Science	Functional	Goal
G-15	2.13 Spare Kit for Vital Functions	Functional	Goal
G-16	2.14 High Cost per Kilogram	Commercial	Obstacle
G-17	2.15 Affordable Price	Commercial	Goal
G-18	2.16 Equipment Failure	Reliability	Obstacle
G-19	2.17 Radiation Damage	Functional	Threat
G-20		Functional	Goal

Reproduced by permission of Telelogic UK Ltd, an IBM company.

Figure C.2: Requirements database underlying a goal model. Each item can be traced to design, tests, definitions etc.

is no consistency checking or data sharing, unless you are ready to do a great deal of programming with scripting languages.

Microsoft Visio (<http://office.microsoft.com/visio>) provides similar but richer drawing facilities. You can make the models quite large, as you can scroll the window. You can attach small pieces of text to Visio diagram objects.

Similar tools are available for **free (open source) download** from <http://www.openoffice.org>. For example, OpenOffice.org Draw provides boxes, connectors, flowchart elements, and so on. The tools are remarkably powerful and easy to use.

If you have the **requirements tool DOORS** available, you are welcome to download the **free Diagrams toolkit** from <http://www.scenarioplus.org.uk>. It was used to create the stakeholder 'onion' models in Chapter 2, the spacecraft goal model in Chapter 3 (its database is shown in Figure C.2) and the rationale models in Chapter 7. Each stakeholder, goal, definition or assumption (etc), is an object in the database, which makes it easy to trace requirements later. There is also a free **Dictionary tool** on the same website that semi-automatically constructs traces between places where terms are used, and their definitions (see Chapter 7).

The advantage of using a graphics editor built directly into a requirements database is that elements such as goals and rationale are modelled not just as objects in a diagram, but as fully-valued records in the database. This means that they can be managed and traced, just like requirements. For example, you can:

- prioritise them;
- track their status through the project;
- ensure they are implemented in the design, and tested;
- ensure that terms used are defined in the project dictionary.

The price for such advantages is discussed below.

Tips for Choosing Tools

- Simpler is often better; anything too technical may make matters worse by limiting communication.
- Think about where you will create diagrams or models, and where you will show them to people:
 - if you need to edit or display diagrams on the move, the tool had better be on your laptop;
 - if you need to email diagrams to people, the output had better be something they can read just by opening the email, or at most by using ordinary office software.
- Sharing should be easy to make discovery work smooth and seamless. Licences that tie you to one machine are less convenient than floating licences (or software that doesn't need licensing, for that matter).

Specialised Modelling Tools

Several kinds of diagram – usually the most complicated ones – are supported by tools made for the purpose. Some are commercial; others are free.

There seems to be quite a weak correlation between the price of modelling tools and their usefulness. For example, for **UML** models (see Chapter 8, Definitions) we use **Enterprise Architect** (<http://www.sparxsystems.com.au/>), which is modestly priced but richly featured. It seems to do everything that the requirements analyst might need from UML, just like products that cost ten times as much. Indeed, you can download free UML modelling tools such as **Visual Paradigm** (<http://www.visual-paradigm.com>).

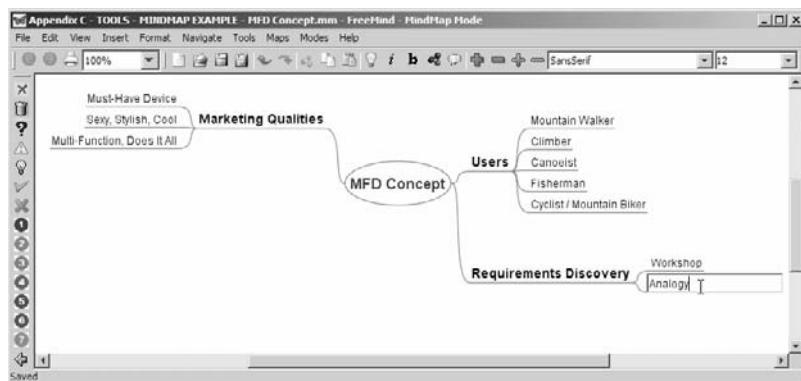


Figure C.3: Brainstorming with FreeMind mind-mapping tool.

Free tools are available for several kinds of model, and more are appearing all the time.

For example, for modelling reasoning (rationale and assumptions), **Compendium** (<http://compendium.open.ac.uk/openlearn/>) is powerful, tailor-able, quick to use, and free. For some purposes, it is all you could wish for. You could probably also use it for modelling goals.

FreeMind (Figure C.3) is a free mind-mapping tool available from SourceForge at <http://freemind.sourceforge.net/wiki/index.php/MainPage>. It is powerful and intuitive to use.

Industry has traditionally been reluctant to use free software because of the lack of product support, should anything go wrong when a project has come to depend on a tool. This problem has largely been solved by the provision of open source licenses, which allow software services companies to charge fees for product maintenance and support, while giving away the core software.

If you need to create specialised kinds of model such as GSN or CAE to express safety or other rationale formally (Chapter 7), you need a tool such as the **Assurance and Safety Case Environment** (ASCE) from Adelard (<http://www.adelard.com>).

If all you want is a diagram, on the other hand, there is no reason to go for a costly tool. The main advantages of the larger products are the abilities to trace between models, and to share data with other tools.

Good reasons for tool support are when projects involve many people, numerous sites, high risk, safety or business-criticality, long product lifetimes, product variants (e.g. for different markets), product lines, and accountability to a regulator.

As with requirements work that you do for your clients, then, the best if rather obvious advice is to work out carefully what you need from your requirements tools, and to obtain tools that do what you need and no more.

Requirements Management Tools

Requirements management (RM) tools provide databases to handle traceability, along with many related facilities to control requirement status, to select requirements by database query, and to manage requirement configuration.

However, the needs of requirements discovery are quite different from those of requirements management. RM is about keeping things neat and tidy, documenting bits of text, tracing one to another, storing baselines and the like. All of this is very helpful for the products of the requirements discovery process, once they exist. But it's relatively little help with the creative task itself.

This book provides 'tools' in the practical sense of 'effective techniques' or 'cognitive tools for problem-solving'. Such 'tools' include types of model, ways of questioning, tips and checklists.

Pure RM tools can be helpful even early in requirements discovery. Tracing issues to goals, for instance, can show where problems are likely to occur. A

Table C.2: Pros and cons of requirements management (RM) tools.

For	Against
<p>Database manages versions and configuration (sets of documents)</p> <p>Traceability between related items in different documents may prevent or solve many requirements issues (e.g. inconsistent handling of dependencies; see Chapter 10, Priorities)</p> <p>Status information (e.g. priority, review status) is easy to keep with each requirement</p> <p>May be able to create models in the same environment</p> <p>May be able to animate/simulate requirements</p> <p>May be able to check for some kinds of errors semi-automatically (e.g. untraced requirements, inconsistencies)</p> <p>Central database permits many people on different sites to work together</p> <p>Project knowledge is held securely</p>	<p>Not much help with requirements creation itself</p> <p>Initial cost may be high</p> <p>Additional training needed for project staff</p> <p>Initial setup, design of data structures, and importing of data can slow down project progress</p> <p>Exporting and formatting may be needed to create documents for review</p> <p>Integration with other tools may be difficult or costly</p> <p>Danger of becoming 'locked in' to a specific vendor</p> <p>Modelling environment, if any, is most likely UML – again, not very helpful for requirements creation</p>

table of requirements can make it easy to see which items have been worked on in detail, and which are still in a raw state. In that sense, you can ‘engineer’ requirements; the systematic application of rules can help to create a usable set of requirements.

Tool vendors are aware that requirements text needs to be related to models. They have responded by linking models, usually UML, to their requirements databases. As the chapters in Part 1 of this book illustrate, many of the models that help to clarify requirements are not supported in the *de facto* industry standard modelling language, UML. This means that RM tools are not as helpful as they could be for requirements creation, but they are certainly better than they were a few years ago.

Adopting an RM tool on your project is quite a serious step. There are definite pros and cons to using RM tools on a project (Table C.2).

In general, the balance shifts steadily towards the need for an RM tool as a project becomes larger, more critical to a business and longer-lived, involves safety, has multiple products (e.g. a product line, an evolving product, variants for different customers, etc) and involves more companies (e.g. in a supply chain).

APPENDIX D

Template

This appendix lists the requirement elements that are described in this book. Your project may not need to populate all of them. Some of the constraints in section 6.1 of the template do not apply to software-only projects, for instance. The list may help you to consider each element in your situation.

Each element is analysed only to a general level, to keep the template small, and to remind you of the need to tailor your approach to suit your project's situation. Ellipses (...) are included as a further reminder that projects vary. Details of the template are discussed in the book, especially in Chapter 2 (Stakeholders) and Chapter 6 (Qualities and Constraints). Versions of the template are also provided on the website <http://www.scenarioplus.org.uk>, with supporting materials, and periodically updated: comments are welcomed.

The template does not require you to work on the elements in the order shown. In general, progress will be broadly in this direction, but you should expect to iterate with both a small-scale inquiry cycle (see Chapter 1) and appropriate life-cycle design, e.g. to reduce risk (see Chapter 15).

The template does not state how the elements should be documented. For example, priorities are likely to become attributes; goals may be documented on a diagram; scenarios may be documented as text, as storyboards, as use cases, and so on. The chapters (indicated in parentheses) suggest suitable ways of documenting each element.

1. Vision or Mission (in Chapter 3)

1.1. *Project Vision/Mission Statement*

2. Stakeholders (Chapter 2)

2.1. *Beneficiaries*

Champion

Sponsor

Functional Beneficiary

Financial Beneficiary

...

2.2. *Operators*

Normal Operator (*there may be many subtypes*)

Maintenance Operator

Support Operator

...

2.3. *Regulators*

Industry Regulator

Government

...

2.4. *Negative Stakeholders*

Competitor

...

2.5. *Other Stakeholders*

Developer

Expert

The Public

...

3. Goals (Chapter 3)

3.1. (*Positive*) *Goals*

3.2. *Obstacles*

3.3. *Threats*

3.4. *Goal Conflicts*

4. Context, Interfaces and Behaviour (Chapter 4)

4.1. *Rich Picture/Context Diagram*

4.2. *Interfaces* (with External Systems)

Incoming, Outgoing or Bidirectional

Physical Connectors

4.3. *Event-Handling Functions* (can be grouped with the Interfaces to which they apply)

'When' Requirements

Condition-Action Tables

Authorisation Rules

Functions, Capabilities

5. Scenarios (Chapter 5)

5.1. (*Positive*) *Scenarios* (alternative approaches)

User Stories

Storyboards
(Operational) Scenarios

- Whole Life
- Day In the Life Of
- Transaction/Operation/Task
- Data Handling (Create-Read-Update-Delete)
- ...

Use Cases

- Functional Goal/Use Case Title
- Primary and other Roles
- Main/Happy Day Scenario
- Variations
- Exceptions
- Preconditions
- Trigger
- Guarantees
 - Success Guarantees
 - Minimal Guarantees
- Stakeholders and Interests
- Local Qualities and Constraints

5.2. Negative Scenarios

- Exceptions (if you are not writing use cases)
- Intentional Threats (misuse cases)
- Unwanted Scenarios (forbidden combinations of actions)

6. Qualities and Constraints (Chapter 6)

6.1. Constraints

Design Constraints

- Use of COTS Products
- Forbidden (e.g. toxic) Materials
- Construction
- Disposability (for recycling)
- ...

Regulations and Standards

Human Factors

User/Operator Population

Console/Cockpit Design

Lighting, Seating, etc

Training

...

Environmental Constraints

Temperature

Humidity, Waterproofing

Vibration, Shockproofing ('shake, rattle and roll')

Electromagnetic Compatibility (EMC)

Allowed Emissions

Required Immunity

...

Physical Constraints

Size and Shape

Weight

Power Consumption

Finish, Colour, and Labelling

...

...

6.2. Development (Process) Qualities

Producibility

Flexibility

Upgradability

Scaleability

Modifiability

...

Testability

...

6.3. Usage (Product) Qualities

Dependability

Availability, Reliability

Maintainability

Safety

Security

Survivability

Performance (may be treated as attribute of functional requirements)

Usability

Interoperability

...

6.4. Programmatic Requirements

Development Requirements

Programming Languages

Documentation

...

Test Requirements

Test Approach

Special to Purpose Test Equipment

Simulators

Trials and Parallel Operations

...

Costs

Timescales

...

7. Rationale (Chapter 7) (may in part be treated as attribute of requirements)

7.1. Supporting Assumptions (Warrant)

7.2. Contradicting Assumptions (Rebuttal)

8. Definitions (Chapter 8)

8.1. Acronyms

8.2. Terms

8.3. Roles

8.4. Data

9. Measurements (Chapter 9) (may be treated as attribute of requirements)

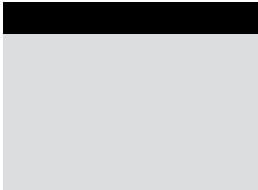
9.1. Acceptance Criteria

9.2. Quality of Service Measures

10. Priorities (Chapter 10) (may be treated as attribute of requirements)

10.1. Input Priorities (Importance to Stakeholders)

10.2. Output Priorities (Agreed for a Development Phase)



Bibliography

This references in this bibliography are also listed in the topical annotated further reading sections at the end of each chapter. The books have been selected for their practical coverage of the chapter topics. Journals and research literature have been avoided as far as possible.

Reviews of many of these books can be found on Ian Alexander's website at www.scenarioplus.org.uk

- Alexander, I. (2004) Negative Scenarios and Misuse Cases. In Alexander I. and Maiden N. *Scenarios, Stories, Use Cases*, Chichester: John Wiley & Sons, Ltd.
- Alexander, I. (2005) A Taxonomy of Stakeholders: Human Roles in System Development, *International Journal of Technology and Human Interaction*, 1 (1), 23–59, http://easyweb.easynet.co.uk/~iany/consultancy/stakeholder_taxonomy/stakeholder_taxonomy.htm.
- Alexander, I. and Maiden, N. (Editors) (2004) *Scenarios, Stories, Use Cases*, Chichester: John Wiley & Sons, Ltd.
- Alexander, I. and Stevens, R. (2002) *Writing Better Requirements*, London: Addison-Wesley.
- Analytic hierarchy process (AHP), http://en.wikipedia.org/wiki/Analytic_hierarchy_process.
- Avizienis, A, Laprie, J-C. and Randell, B. (2001) *Fundamental Concepts of Dependability*, Research Report N01145, LAAS-CNRS, April 2001.
- Beyer, H. and Holtzblatt, K. (1998) *Contextual Design, Defining Customer-Centered Systems*, London: Morgan Kaufmann.
- Boehm, B. et al, *WinWin Spiral Model and Groupware Support System*, <http://sunset.usc.edu/research/WINWIN/index.html>.
- Boston Matrix http://www.marketingteacher.com/Lessons/lesson_boston_matrix.htm.

- Bruegge, B. et al, *Software Cinema*, 2004, <http://wwwbruegge.in.tum.de/publications/includes/pub/bruegge2004softwarecinema/bruegge2004softwarecinema.pdf>.
- Burge, J.E., Carroll, J.M., McCall, R. and Mistrík, I. (2008) *Rationale-Based Software Engineering*, Berlin: Springer.
- Buzan, T. (2002) *How to Mind Map*, London: Thorsons.
- Carroll, J.M. (Ed), (1995) *Scenario-Based Design: Envisioning Work and Technology in System Development*, New York: John Wiley & Sons, Inc.
- Checkland, P. and Scholes, J. (1990; 1999) *Soft Systems Methodology in Action*, Chichester: John Wiley & Sons, Ltd.
- Cockburn, A. (2001) *Writing Effective Use Cases*, Boston: Addison-Wesley.
- Cockburn, A. (2006) *Agile Software Development*, 2nd Edition, Boston: Addison-Wesley.
- Cohn, M. (2004) *User Stories Applied: For Agile Software Development*, Boston: Addison-Wesley.
- Crabtree, A. (2003) *Designing Collaborative Systems, A Practical Guide to Ethnography*, London: Springer.
- Davis, A.M. (2005) *Just Enough Requirements Management*, New York: Dorset House.
- DeMarco, T., Hruschka, P., Lister, T., McMenamin, S., Robertson, J. and Robertson, S. (2008) *Adrenaline Junkies and Template Zombies: Understanding Patterns of Project Behavior*, New York: Dorset House.
- Dewar, J. (2002) *Assumption-Based Planning*, Cambridge: Cambridge University Press.
- Dieste, O., Juristo, N. and Shull, F. (2008) Understanding the Customer: What Do We Know about Requirements Elicitation?, *IEEE Software* March/April, 25(2) 11–13.
- Dreyfuss, H. (1955) *Designing for People*, New York: Simon and Schuster.
- Dytham, C. (2003) *Choosing and Using Statistics*, 2nd Edition, Malden: Blackwell.
- Eastaway, R. and Wyndham, J. (2005) *Why Do Buses Come in Threes? The Hidden Mathematics of Everyday Life*, London: Robson Books.
- Farncombe, A. (2004) Project Stories: Combining Life-Cycle Process Models. In Alexander, I. and Maiden, N. *Scenarios, Stories, Use Cases*, Chichester: John Wiley & Sons, Ltd.
- Fowler, M. (2004) *UML Distilled*, 3rd Edition, Boston: Addison-Wesley.
- Gause, D.C. and Weinberg, G.M. (1989) *Exploring Requirements: Quality Before Design*, New York: Dorset House.
- Goldsmith, R.F. (2004) *Discovering REAL Business Requirements for Software Project Success*, Boston: Artech House.
- Gottesdiener, E. (2002) *Requirements by Collaboration: Workshops for Defining Needs*, Boston: Addison-Wesley.
- Heron, J. (1996) *Co-operative Inquiry: Research into the Human Condition*, London: Sage.

- Hooks, I.F. and Farry, K.A. (2001) *Customer-Centered Products: Creating Successful Products Through Smart Requirements Management*, New York: Amacom.
- Hull, E., Jackson, K. and Dick, J. (2005) *Requirements Engineering*, 2nd Edition, London: Springer.
- i* (goal modelling notation), <http://www.cs.toronto.edu/km/istar/>.
- Jackson, M. (1995) *Software Requirements and Specifications, a lexicon of practice, principles and prejudices*, Harlow: Addison-Wesley.
- Jirotka, M. and Luff, P. (2006) Supporting Requirements with Video-Based Analysis, *IEEE Software May/June*, 23(3) 42–44.
- Kelly, T. and Weaver, R. (2004) *The Goal Structuring Notation – A Safety Argument Notation*, <http://www-users.cs.york.ac.uk/~tpk/dsn2004.pdf>.
- Kirschner, P.A., Buckingham Shum, S.J. and Carr, C.S. (Eds) (2003) *Visualizing Argumentation*, London: Springer.
- Kotonya, G. and Sommerville, I. (1998) *Requirements Engineering, Processes and Techniques*, Chichester: John Wiley & Sons, Ltd.
- Kuniavsky, M. (2003) *Observing the User Experience: A Practitioner's Guide to User Research*, San Francisco: Morgan Kaufmann.
- Lauesen, S. (2002) *Software Requirements, Styles and Techniques*, Harlow: Addison-Wesley.
- Lauesen, S. (2004) *User Interface Design, A Software Engineering Perspective*, Harlow: Addison-Wesley.
- Lauesen, S. (2007) *Guide to Requirements SL-07: Template with Examples*, Copenhagen: Lauesen Publishing.
- Lavi, J.Z. and Kudish, J. (2005) *Systems Modeling & Requirements Specification Using ECSAM: An Analysis Method for Embedded and Computer-Based Systems*, New York: Dorset House.
- Levine, R., Locke, C., Searls, D. and Weinberger, D. (2000) *The Cluetrain Manifesto, The End of Business as Usual*, London: FT.com.
- Maciaszek, L.A. (2005) *Requirements Analysis and System Design*, Harlow: Addison-Wesley.
- MacLean, A. and McKerlie, D. (1995) Design Space Analysis and Use Representations. In Carroll, J.M. (Ed) *Scenario-Based Design: Envisioning Work and Technology in System Development*, New York: John Wiley & Sons, Inc.
- Maier, M.W. and Rechtin, E. (2000) *The Art of Systems Architecting*, Boca Raton: CRC Press.
- O'Leary, D.E. (2008) Wikis: 'From Each According To His Knowledge', *Computer* 41 (2) 34–41.
- Parnas, D.L. and Clements, P.C. (2001) A Rational Design Process: How and Why to Fake IT. In Hoffman, D.M. and Weiss, D.M. *Software Fundamentals, collected papers by David L. Parnas*, Boston: Addison-Wesley.
- Pinker, S. (2007) *The Stuff of Thought*, London: Allen Lane.
- Polanyi, M. (1966) *The Tacit Dimension*, Gloucester, Mass: Doubleday.
- Quality function deployment (QFD), http://en.wikipedia.org/wiki/Quality_function_deployment.

- Rechtin, E. (1990) *Systems Architecting, Creating and Building Complex Systems*, Upper Saddle River, NJ: Prentice Hall.
- Robertson, S. and Robertson, J. (2004) *Requirements-Led Project Management: Discovering David's Slingshot*, Boston: Addison-Wesley.
- Robertson, S. and Robertson, J. (2006) *Mastering the Requirements Process*, 2nd Edition, Upper Saddle River, NJ: Addison Wesley.
- Schank, R.C. (1990) *Tell Me a Story: Narrative and Intelligence*, Evanston, Illinois: Northwestern University Press.
- Schrage, M. (1999) *Serious Play: How the World's Best Companies Simulate to Innovate*, Boston: Harvard Business School Press.
- Simon, H.A. (1996) *The Sciences of the Artificial*, 3rd edition, Cambridge, Mass: MIT Press.
- Smith, L.I. (2002) *A Tutorial on Principal Components Analysis*, http://csnet.otago.ac.nz/cosc453/student_tutorials/principal_components.pdf.
- Snyder, C. (2003) *Paper Prototyping: The Fast and Easy Way to Design and Refine User Interfaces*, San Francisco: Morgan Kaufmann.
- Sommerville, I. (2006) *Software Engineering*, 8th Edition, Harlow: Addison-Wesley.
- Stevens, R., Brook, P., Jackson, K. and Arnold, S. (1998) *Systems Engineering, Coping with Complexity*, London: Prentice Hall.
- Sutcliffe, A. (2002) *User-Centred Requirements Engineering, Theory and Practice*, London: Springer.
- Suzuki, S. (1970) *Zen Mind, Beginner's Mind, Informal talks on Zen meditation and practice*, New York and Tokyo: Weatherhill.
- Tanenbaum, A.S. (2002) *Computer Networks*, 4th Edition, Upper Saddle River, NJ: Prentice Hall.
- Thiagarajan, S. (2003) *Design Your Own Games and Activities: Thiagi's Templates for Performance Improvement*, San Francisco, CA: Pfeiffer.
- Toulmin, S. (1958) *The Uses of Argument*, Cambridge: Cambridge University Press.
- Vincenti, W.G. (1990) *What Engineers Know and How They Know It: Analytical Studies from Aeronautical History*, Baltimore and London: Johns Hopkins.
- Wiegers, K.E. (2003) *Software Requirements*, 2nd Edition, Redmond: Microsoft Press.
- Wiley, B. (1999) *Essential System Requirements: A Practical Guide to Event-Driven Methods*, Reading, Mass: Addison-Wesley.
- Young, R.R. (2001) *Effective Requirements Practices*, Boston: Addison-Wesley.

Glossary

Terms in **bold** are cross-references to other glossary entries.

Acceptance criterion (plural is criteria)	Means of measurement ; a condition under which a requirement is to be accepted as met when tested or otherwise verified before release . Contrast with quality of service .
Actor	UML term roughly equivalent to operational role . UML does not cover nonoperational roles. However: <ol style="list-style-type: none">1. actors need not be human, so a product, capable of making decisions, with an interface to our system is in UML terms an actor;2. actor normally only covers roles that directly interact with a product, whereas in a system, human actors often also interact with other human operators to provide the wanted results, and it may not be known whether such interactions will be automated.
Assumption	Condition (of something outside the product or system) that is taken to be true, and relied on by the project, e.g. as part of the rationale for the requirements, but cannot be controlled. Contrast with requirement .

Attribute	Component of a requirement or requirement element , describing e.g. its priority , status , rationale . A field on a data entry form or in a database often corresponds to a requirement attribute.
Baseline	Published version of a project document or set of documents, whether for stakeholders to comment on or for developers to work from. As such, a baseline is always treated as read-only; agreed changes are incorporated into a subsequent (not yet baselined) version.
Beneficiary	Stakeholder role involving direct or indirect benefit of any kind from the product or system . For example, a financial beneficiary expects to gain money from the system's operations. Subclasses include: financial; functional ; political; social.
Business event	An event in the world that a project decides is in scope .
Business objective	A (brief, high-level) statement of a project 's overall purpose. May be analysed into detailed and more readily realisable goals .
Business requirement	A requirement that comes from stakeholders and states a result that is wanted in the world, without making assumptions about how that will be delivered.
Cherry stones	Method of triage (named after an old lover's game) with values like {this phase, next phase, sometime, never}.
Constraint	A requirement that places a specific limitation on: <ol style="list-style-type: none"> 1. how a product may be constructed, e.g. on allowable power consumption (see Chapter 6, Qualities and Constraints); 2. how a software product may behave, e.g. that data in a field may only take certain values, perhaps governed by data in another field (see Chapter 8, Definitions).
Criterion (plural is criteria)	A measurable goal against which design options can be compared in trade-off analysis.

CRUD	Create – Read – Update – Delete: typical scenarios for handling data (often called ‘housekeeping’). Useful as a completeness check during validation .
Definition	A statement of the meaning of a dictionary term (normally an item of data or a role) by reference to other dictionary terms. Contrast with designation .
Dependability	A major subclass of quality requirements , concerned with how far people may rely on a product (generally one that operates in real time) for continued correct operation. Subclasses include: availability; reliability; safety; security; survivability.
Design	<p>1. The process of stating how a product is to be constructed, as opposed to requirements (which state what and why).</p> <p>2. The documented output of that process; for hardware/software systems, this is also called an ‘architecture’.</p> <p>The design of a containing system imposes interface requirements on a contained subsystem.</p>
Design element	A part of a design . See also feature . Contrast with requirements element .
Designation	A description of the meaning of a glossary term in the project dictionary by reference to things in the outside world. Contrast with definition .
Development project	= project .
Dictionary	= project dictionary .
DILO	‘A Day In the Life Of’, a standard scenario pattern. See also CRUD .
Domain	A realm of business discourse, e.g. air traffic control, railway, broadcasting, banking, telecommunications, retail.
Event	Something that happens of significance to the product , at one of its interfaces . Subtypes are business event , (internal) ‘system’ event.

Exception	<p>1. (= exception event) an event that interferes with a scenario (i.e. progress towards a functional goal). 2. (= exception scenario) a scenario describing how to handle an exception event; see use case.</p>
Feature	A functional design element visible at business level, i.e. from the outside of a product . In an ongoing programme , features are grouped into releases .
Financial (stakeholder)	Human role for stakeholders deriving money from the success of a product or system , e.g. shareholders, company directors.
Function, -al	(of a requirement) calling for a specific capability, i.e. a defined piece of behaviour in response to a specific stimulus, from the product or system . (of a beneficiary) making use of such capabilities. Compare normal operator, user .
Global	(of an NFR) applying to the whole system or product . Opposite of local .
Glossary	The informal part of a project dictionary , designating terms used in the world outside the product .
Goal	<p>1. (Functional or nonfunctional) statement (a requirement element) of a stakeholder's desired target for a project. May not be completely realisable or verifiable. Nonfunctional goals can often be decomposed into functional goals. Compare obstacle.</p> <p>2. Requirement that has not yet received an output priority; even if strongly desired by stakeholders, its inclusion in a (phase of a) project is uncertain.</p>
'Housekeeping'	Colloquial engineering term for managing data. See CRUD .
'-ilities'	Colloquial engineering term for qualities e.g. reliability.
Information model	Coarse-grained model of the information structure of a product development project, suitable for

	designing a requirements database. The building blocks of the model are the types of requirement element .
Input priority	Priority of a requirement in the opinion of stakeholders , i.e. how much they want it. Compare output priority .
Interface	(Requirement for) a system or product's ability to interchange data or anything else with any other system, or to have a connector of a specific design so as to enable such interchange. Interface requirements are thus properly defined in complete design detail (down to bits and bytes). Interfaces are often standardised and can thus be specified simply by calling out a specific section or clause of a standard .
IPT	Integrated project team; a group of people including representative stakeholders who are not themselves developers, assembled to enable effective development of a product .
Issue	A matter arising from the requirements that needs to be understood and resolved before the requirements can be released .
Justification	Rationale (for a requirement) written as a brief text attribute .
Key (requirement)	Approach to setting priorities in which only a small number of Requirements (say, seven) may be designated as 'key', i.e. absolutely necessary.
Local	(of an NFR) applying only within a limited scope, e.g. to a single function or use case. Opposite of global .
Measurement	<ol style="list-style-type: none"> 1. For a product that will be verified, then released: acceptance criterion. 2. For a service that will be delivered continuously: quality of service.
Mission	A short statement of the purpose and scope of a product development project.
Misuse case	Hostile functional goal representing an intentional threat (desired by a negative stakeholder).

Multifunctional device (MFD)	Fictional handheld product-under-development for outdoor leisure purposes used in examples and exercises in this book.
Negative stakeholder	Stakeholder whose goals appear as obstacles to the project.
Nonfunctional requirement (NFR)	Any requirement that is not a function . Includes qualities and constraints .
Normal operator	Human role that interacts with the product to deliver results (to functional beneficiaries). Compare user .
Obstacle	Something (intentional or not) that obstructs progress towards a goal . Subtypes include exception events and threats .
Operational scenario	Type of scenario , usually told as a simple story or sequence of steps , describing how human operators (possibly playing many roles) interact with each other and with products in a system to achieve a goal .
Operator; operational (stakeholder)	Human role for stakeholders actively involved in making the system work, often but not exclusively by direct use of the product . Subclasses include normal operator , maintenance operator.
Optioneering	= trade-off analysis.
Output priority	Priority of a requirement with respect to the chosen design after optioneering ; can be expressed in terms of time, i.e. which release the requirement will be in. Compare input priority .
Political (stakeholder)	Human role for stakeholders deriving any kind of power or prestige from a system or product .
Prioritisation	Process of assigning priorities to requirements or features , with the purpose of triage .
Priority	The relative importance to stakeholders (input priority) or agreed order of implementation (output priority) of a requirement or feature .
Product	The thing under development, whether software, hardware (computational or otherwise) or a combination; and whether for sale as a commercial item or part of one, or developed in-house. Forms part of a system (excluding the system's human components, etc). Compare service .

Product line	Set of products (from a single manufacturer) sharing a core of reused requirements , and possibly also sharing some standardised hardware or software components.
Product manager	Human role in a product development organisation, responsible for managing a product or product line through its various releases , on behalf of the organisation, and as surrogate for (mass market) users .
Product specification	(A set of) requirements specifying a product after optioneering in sufficient detail for it to be developed. A product specification may refer to any design elements that have been agreed in the optioneering process. Contrast with stakeholder requirements .
Programme	Set of related projects , often sharing requirements and team members. When a programme consists of a sequence of developments of the same product , resulting in successive releases , the project-like developments are described as 'programme phases'.
Programmatic	(Issue , requirement) concerning team management, organisation, cost, timescale, etc.
Project	Time-limited undertaking with a single purpose: to create the new thing – the product or service – named in its mission .
Project dictionary	Set of interlocking definitions of terms used within the project. At least three kinds of term may be included: glossary designations ; role definitions; and (hierarchical) data definitions. Acronyms may be included, but should be defined as well as expanded into full names of terms. Data items may be further subtyped into messages etc.
Qualities	Requirements stating the manner ('safely', etc) in which a product to operate, as opposed to its functions . Popularly called the '-ilities' for the names of many subclasses. Subclasses include: dependability ; maintainability; verifiability; usability; safety; security.
Quality of service (QoS)	Measurement , made continuously or at regular intervals, of a service to determine its acceptability

	for contractual and financial purposes. Contrast with acceptance criteria .
Questions, options, criteria (QOC)	An approach to trade-offs using a simple diagram of the traces between a question, a set of (design) options, and a set of criteria (based on requirements).
Rationale	Definition of the reason or purpose of requirements , often by traces to underlying goals or assumptions . See also justification .
Rebuttal	Argument against a conclusion, forming part of a rationale . Opposite of warrant .
Regulations	Documents with legal force, created by legislation, listing requirements on all systems , products , services and businesses of stated types. Regulations are applicable whether or not they are mentioned in a project's requirements. Compare standards .
Regulator	Stakeholder role responsible for controlling some aspect of the system , often with legal authority.
Release	A (numbered) version of a product issued to customers, often associated with a programme of development of successively more features .
Requirement	<p>1. (Narrow sense) Verifiable condition of a (class of) product or system, initially false, that must be made to become true by the development project. Traditionally written in a style such as 'The system shall do XYZ', for inclusion in a commercial contract. Contrast with assumption; goal. Subclasses include: function; quality; constraint; interface.</p> <p>2. (Broad sense) Any requirement element; or, as 'the requirements', equivalent to 'a specification'.</p>
Requirement element	A work product of the process of discovering requirements . Subclasses include: stakeholder analysis ; goals ; scenarios ; rationale ; assumptions ; definitions ; priorities ; measurements ; qualities ; constraints . See also information model .

Requirements definition	= requirements discovery .
Requirements discovery (RD)	The process of finding out the requirements for a project , by working together with the stakeholders . Contrast with requirements management .
Requirements management (RM)	The process of tracking requirement status, priority , progress and change throughout the development life cycle. Contrast with requirements discovery .
Reuse	Adoption of an existing requirement on a project. See also standards , regulations , product line .
Risk	Anything that might interfere with a project's ability to create a successful product or system .
Role	<p>1. (Broad sense) Part played by any class of stakeholders with respect to a product, e.g. its operator, regulator.</p> <p>2. (Narrow sense) Part played by any class of operational stakeholders, i.e. interacting with other stakeholders or with products within a system to deliver results as stated in requirements. E.g. normal operator; maintenance operator.</p> <p>Compare actor.</p>
Scenario	Time-sequence of steps to achieve a functional goal . Synonyms are: course; flow; path; sequence; or, informally, 'story'. See also operational scenario , use case .
Scope	The envelope or boundary of a system or product , defined ultimately by the complete set of requirements , but initially by an informal statement of objectives and then by a context diagram or its equivalent.
Service	The provision of some desired behaviour to a functional beneficiary by a system (possibly containing one or more products , possibly containing humans), usually in return for payment. See also quality of service .
Soft Systems Methodology (SSM)	Approach pioneered by Peter Checkland for understanding the ill-defined behaviour of social and socio-technical systems .

Specification	Description of a product or a part of one, defining what is to be built. Necessarily assumes some knowledge of the chosen design. Contrast with stakeholder requirements .
Stakeholder	Human or other legal entity (company, etc) playing a system role and thereby having a valid interest in the development of the system or product . Subclasses include: beneficiaries ; operator ; regulator ; negative stakeholder .
Stakeholder requirements	Set of requirements stating desired results in the world, before optioneering , i.e. without assuming knowledge of any future product's design. Contrast with product specification .
Standards	Documents listing requirements intended to be reused on all systems , products , services or businesses of stated types, with a degree of force ranging from advisory (e.g. industry guidelines, best practice, yellow books) to mandatory (national and international standards). Standards often become applicable only when they are explicitly called up in a project's requirements. Compare regulations .
Step	Single functional activity undertaken either by an operator or by a product , and, in a time-sequence with other steps, forming a scenario .
Surrogate (role)	Stakeholder role that exists to represent or serve the interests of other groups of stakeholders. E.g. Marketing represents product customers.
System	The network of operational roles , procedures, product(s) and interfaces that together can deliver the wanted results to functional beneficiaries .
System requirements	[In common usage] Product specification (note that this usage conflicts with the definition of system). [Contrasted in common usage with user requirements].
Test case	Information structure combining functional goals and scenarios , used to demonstrate by test that a scenario or requirement has been met by a product or system .

Threat	Goal desired by a hostile stakeholder , and which endangers a goal of the project. Compare obstacle .
Trace	A navigable reference between two project information items, e.g. a database link, hyperlink, or paragraph reference. See traceability .
Traceability	The ability to navigate between related information elements, possibly of different kinds (requirements , goals and other requirement elements , design elements, tests, etc) for validation and other purposes including verification and product release planning. Traces are a major means of providing rationale .
Trade-off (analysis)	Process of finding the design option(s) which best meet(s) the requirements . In turn, the chosen design determines which of the requirements and their measurements can be met. Output priorities generally depend on the results of trade-off analysis.
Triage	Process of sorting requirements into groups, such as {obviously needed, obviously not needed, and to be prioritised in more detail}. Makes use of prioritisation , trade-off .
Unified Modeling Language (UML)	The de facto industry standard language for modelling data and software behaviour. UML was devised in the context of software systems, but has steadily spread into other areas. Specialised versions of it have been created to encourage its use in other areas: e.g. SysML for systems engineering. However, at present, UML does not cover many requirement elements . See also constraint .
Use case	Information structure combining functional goals , roles , (functional requirements represented as) scenarios , exceptions and associated requirements including pre- and post-conditions and local NFRs .
User	Hybrid role combining functional beneficiary , normal operator and possibly mass-market consumer as well.

User requirements	[In common usage; obsolete term for] business requirements . [Contrasted in common usage with system requirements .]
Validate; validation	Checking that a requirement correctly reflects the intentions of the stakeholders , and is realistic given the resources of the development project and the laws of nature. See also verification . Note that the terms are sometimes interchanged, but the activities remain distinct.
Verifiable	(Quality of a requirement) Capable of verification . Compare goal .
Verify; verification	Checking that a product or system meets the requirements placed upon it, whether by test, analysis, or other means. See also validation . Note that the terms are sometimes interchanged, but the activities remain distinct.
Warrant	Argument in favour of a conclusion, forming part of a rationale . Opposite of rebuttal .

Index

Note: Page references in **bold** indicate tables and those in *italics* indicate figures.

- acceptance criteria, 219, 378
 - for behavioural requirements, 212–216
 - for constraints, 218–219
 - discovery and documentation of, 211–221
 - for functions, 212, 213
 - for qualities, 216–218
 - tips for, 223
 - validation of, 222–223
- acceptance tests, in agile software development, 213
- acronyms, 191, 195
- activities, for workshops, 294–297
- agile methods, 19, 20, 115, 133, **214**
- airport terminal transport facilities, affect of design options on requirements, **348**
- Alexander, Ian, 17
- algorithms, analysis of, 201
- Ambler, Scott, 347
- analogy, requirements from analogous products, 336–337
- analysis of algorithms, 201
- cost-benefit, 250
- of traceability, 184–186
- analytic hierarchical process (AHP), 356
- Anytown, solution to traffic problem, PCA example, 360–366
- apprenticeships, 276–277
- archaeology, 334–335, 337
- Assumption-Based Planning* (Dewar), 164
- assumptions, 161–188, 296
 - discovery of, 163–169
 - documentation of, **394**
 - inferred, 167
 - reasons for, 164–165
 - validation of, 183–187
- audio recording, documentation of observations, 278–279
- authorisation rules, **90**
- automotive electronic control units (ECUs), 320
- availability, 148–149, 150–151
- Beatty, Joy, 293
- Beck, Kent, 319
- beginner’s mind, 273–274, 277
- behavioural requirements, acceptance criteria for, 212–216
- beneficiaries, 30–32, 38
 - financial, 31, **48**
 - functional, 31, **48**

- beneficiaries, (*continued*)
in a transport project, 31
'best practice' standards, 32, 33
Beyer, Hugh, 7, 13
bias, process, 375
Boston Matrix, 250
boundaries, validation of choice
of, 86–87
brainstorming, 295
breakdown periods, 150
British Standards Institution (BSI), 32
brown-field requirements (changes to
existing systems), 330–335
Bruegge, Bernd, 279
business analysts, 13
business events, 89
business goals, 178–180
business needs, approaches for
discovery of, 391–392
business requirements, 11, 12
business rules, 203

capability, 346
car, telematics system, 212
card sorting, for discovering input
priorities, 240–241
carpeting service, 224–225
case studies
retail IT project, 377–379
transport planning, 379–381
centralisation, 378
ceramics, sustainability of, 144
champions, 34–35
change management process, 333
Checkland, Peter, 13, 77
checklists
discovery of qualities and
constraints, 136–141
hardware and software
system, 136–138
software projects, 139–140
for validation of qualities and
constraints, 158
classical reasoning, 177
clustering goals, 295
Cockburn, Alistair, 120, 124
Cohn, Mike, 20, 213, 214

collaboration at a distance, 311
combination workshops, 297
commercial off-the-shelf (COTS)
products, 143, 350
commercial threats, 67
communication trust, 303
company standards, 32
competency, 303
computer-aided design (CAD)
products, 336
computers
using in documenting
observations, 278
using in interviews, 271–272
concept definition, dominant
requirements elements and
discovery methods, 377
condition-action tables, 90
confidentiality, 152, 285
conflicts, 264, 303
constraints, 296, 382
acceptance criteria for, 218–219
analysis of stakeholders to
discover, 136
as data, 202–204
definition of, 132
design constraints, 142
discovery of, 133–141
documentation of, 141–157, 393
environmental, 144–145
human factors, 144
physical, 145–146
regulations and standards, 142–143
that matter to people, 133
with tolerances, 146
using checklists to discover, 136–141
using goals to discover, 134–136
validation of, 157–159
consultants, 38
context 75–96
diagrams, 87, 92–93, 295, 393
in retail IT project case study, 378
contextual design, 13
Contextual Design (Beyer and
Holtzblatt), 7
contractual trust, 303
corporate acronyms, 191

- corporate glossary, 191
 correlation matrix, 363
 cost-benefit analysis, 250
 create-read-update-delete (CRUD)
 scenarios, 106, 123
 creep, 11, 76, 87
 criteria, qualitative, 361
 culture of an organisation, 196
 current measure(s), 12
 customers, 58, 334
- data, constraints as, 202–204
 data definitions, 201, 202
 data dictionary, 201
 data models, 201, 202, 205, **392**
 Davis, Al, 239
 day in the life of (DILO) scenarios, 105
 decisions, 173, 186
 PCA and making, 362–366
 retail IT project case study, 378
 definitions, 192, 196–197
 discovery of, 190–206
 documentation of, **394**
 in retail IT project case study, 378
 of ‘service’ in different domains, 194
 demonstrations
 face-to-face workshops, **325**
 of a ‘Human CPU’ prototype, 324
 of prototypes, 322–329
 remote, 324–325
 ‘screens-only mockup’, 326–327
 of software, 325–326
 dependability, 147–148
 dependability goals, 135
 dependencies, of requirements, 247–250
 design
 affect of design options on
 requirements, **348**
 constraints, 142
 discussion of options, 352–353
 options, 345
 questions, options, criteria
 (QOC), 353
 selection, 352–360
 weighting approaches, 353–360
 designations, 197–198
 developers, 29, 38
- development life cycle, 376
 development projects, 5, 8
 development qualities, 146–147
 flexibility, 146–147
 producibility, 146
 testability, 147
 devices, rationalising multifunction
 specifications, 166–168
 Dewar, James, 164, 165
 diagrams
 drawing by hand, 299
 drawing in real-time, 268–271
 . see also context diagrams; goal
 diagrams; strategic dependency
 diagrams; strategic rationale
 diagrams; traditional context
 diagrams; use case diagrams
 dictionary, 190, 198
 digital cameras, recording
 with, 300–301
 discovery, 7–9
 activities after discovering
 requirements, 374–375
 of business needs, **391–392**
 of definitions, 190–206
 of dominant requirements
 elements, **377**
 of input priorities, 237–241
 of missing functions, 94
 of negative scenarios, 107–114
 of output priorities, 243–250
 of qualities and constraints, 133–141
 as a search, 18
 discovery contexts, **383**
 discovery cycle, 18–20
 distance, groupware and working at
 a, 310–312
 document templates, 385
 documentation
 of input priorities, 241–242
 of interfaces, 84–86
 of interviews, 268–272
 of observations, 277–279
 of output priorities, 251–253
 of performance, **388**
 of qualities and constraints, 141–157
 of rationale, 169–183

- documentation (*continued*)
of requirement elements, 393–394
of scenarios, 114–123
of use cases, 122
domains, 197
duplicates, 172
- Effective Requirements Practices*
(Young), 382
- electromagnetic compatibility (EMC), 145
- environmental benefits, 355
- environmental constraints, 144–145
- ‘equivalence’, 192
- ethnography, 275
- event-driven analysis, 391
- event-handling functions, 89–92
- event-handling requirements, 89–90
- event-triggered business rules, 89, 91
- events
exception, 107–111
scope as a list of, 87–89
validation of, 93–94
- Excel, 171
- existing products and systems, 330–335
- experts, 38, 48
- expression, of event-handling
functions, 89–92
- external events, 88
- facilitation teams, 300
- failure mode evaluation and criticality
analysis (FMECA) tests, 148
- feature interactions, 68
- features, development of, 249
- fieldwork, 275
- financial beneficiaries, 31, 48
- Financial Services Authority (FSA), 32
- financial software, dominant
requirements elements and
discovery methods, 377
- flexibility, 146–147
- flipcharts, 300
- flowcharts, 105
- food
QoS measurements for preparation
services, 228–230
- scenarios and commercial services for
preparation of, 227
- Fowler, Martin, 271
- functional beneficiaries, 31, 48
- ‘functional equivalence’, 192
- functional goals, 58–59
- functional NFRs, 140–141
- functional specifications, 11
- functions, 17, 52, 66
acceptance criteria, 212, 213
discovery of missing, 94
organisation of, 386–387
in software, 66
and their performance, 212–214
- games, workshop, 293–294
- glass, sustainability of, 144
- glossary, corporate, 191
- goal diagrams, 69
- goal measure(s), 12
- goal models, 295, 391
- goal structure notation (GSN), 182–183
- goals, 51–74
achievement of, 158
analysis of dependability, 135
assumptions, 382
clustering, 295
conflicts, 62–63, 71–72
definition of, 52
differences between requirements
and, 54
discovery of, 52–54, 63–65
documentation of, 68–70, 393
functional, 58–59
of group work, 284–285
inferred (assumed), 167
negative side of, 65–68
for a new (EHR)-system, 180
obstacles, 66–67
quality, 58–59, 136
for a restaurant, 57–59
in retail IT project case study, 378
for a spacecraft, 54–58
stakeholders, 237
in theory and practice, 6
using to discover qualities and
constraints, 134–136

- validation of, 71–73
- workshop, 64–65
- Gottesdiener, Ellen, 296, 302
- GPS (Global Positioning System)
 - specification, 84
- ‘green’ car, design options affect on requirements, 348
- group media, 285, 305–314
- group work
 - discovery of requirements, 284–285
 - goals, 284–285
 - mediation of, 285
 - obstacles, 284, 285
- groups
 - large, 298
 - requirements from, 283–316
- groupware, 285
 - capabilities of, 312
 - pros and cons of, 313
 - and working at a distance, 310–312
- guarantees, 215
- ‘guidance’ standards, 32, 33
- Guide to Requirements* (Lauesen), 150, 178
- handheld consumer electronics devices, 348
- hard systems engineering, 79, 87–95
- hardware dependability, 147
- Holtzblatt, Karen, 7, 13
- homonyms, 193–194
- hospitals, cleaning services choices, 226
- hostile stakeholders, intentional threats, 111–113
- Houdek, Frank, 300, 320, 324
- House of Quality, 357, 358–359
- i* goal modelling, 59–60, 74
 - strategic dependency diagram, 60
 - strategic rationale diagram, 60
- IEEE 830 standard, 138
- if ... then business rules, 91
- image processing, 301
- improvisational theatre techniques (improv), 287, 289
- index cards, 115, 116
- individuals, requirements from, 259–282
- influences model, 294
- informal conversations, 260
- information models, 176
- infrastructural dependency, 248
- input priorities, 236, 237–243, 357
 - discovery of, 237–241
 - documentation of, 241–242
 - validation of, 242–243
- inquiry cycles, 18–19
- inside-out use cases, 123
- Institution of Engineering and Technology (IET), 32
- integrated logistics support (ILS), 149
- Integrated Project Team (IPT), 45
- intentional threats, from hostile stakeholders, 111–113
- interactive products, 111
- interactive software, dominant requirements and discovery methods, 377
- interface connectors, 140
- interface definitions, 201
- interfaces, 17, 33
 - documentation of, 84–86, 393
 - handling different types of, 85
 - identification of, 83–84
 - incoming, 86
 - outgoing, 86
 - physical connectors, 86
 - between printer and computer, 83
 - validation of, 93–94
- interfacing roles, 38, 48
- internal auditors, 199, 200
- international standards, 32
- International Standards Organisation (ISO), 32
- internationalisation requirements, 156
- interoperability, 156–157
- interviewees, 264–265
- interviewers, number of, 264–265
- interviews, 63, 64, 260, 261–274
 - advantages of, 262
 - disadvantages of, 262–264
 - for discovery of scenarios, 99–101
 - documentation of, 268–272
 - of employees with their boss, 265
 - getting second opinions, 273
 - models used in, 270

- interviews, (*continued*)
note-taking, 268
planning, 261–268
questions to ask in, 267–268
recording, 272
time needed for a campaign of, 266
tips for interviewing, 269
using computers in, 271–272
validation of findings, 273–274
- ISO/IEC 9126 standard, 138
- issues, 172
retail IT project case study, 378
- iteration, of requirements and design, 346–347
- Jackson, Michael, 196, 270
- Jackson Structured Design* (Jackson), 196
- Jackson’s Principle of Commensurate Care, 109–111
- jargon, 197
- Just Enough Requirements Management* (Davis), 239
- justification text field, 171–172
- knowledge, showing, 263
- Lauesen, Soren, 150, 178, 213
- lending library, 336
- life history, of requirements, 245–246, 247
- low environmental impact car, design options affect on requirements, 348
- machine action lists, 102–103
- Mahaux, Martin, 288–289
- maintainability, 149
- maintenance interface, 84
- maintenance roles, 48
- maintenance technician, 199, 200
- manufacturers, 48
- marketing roles, 48
- matrix techniques, 241
- mean time between failures (MTBF), 148
- mean time to repair (MTTR), 148
- means of delivery, 226, 228
- measurement scales, 361
- measurements, 209–234, 382, 389, 392
documentation of, 394
- of quality of service (QoS), 223–230
in retail IT project case study, 378
- mediation, of group work, 285
- meetings, 286
- Meyer, Bertrand, 214
- military systems, threats to, 34
- minimal guarantees, 215
- missions, 53
- misuse cases, 111–113
. *see also* use cases
- modelling tools, 308–309, 313
- moderators, 300
- MoSCoW, 242
- MTBF (Mean Time Between Failures), 217
- N² matrix, 68
- naïve reuse, 337–338
- naïve weighting, 354
- names, 196
- national standards, 32
- navigation device actions, 103
- negative scenarios, 285
discovery of, 107–114
exceptions, 107–111
validation of QoS with, 232
- negative stakeholders, 33–34
- nonfunctional goals, for multi-function device, 134
- nonfunctional requirements (NFRs), 132, 137
- notebooks, 278
- objectives, 53
- observations, 260, 261
benefits of, 275
for discovery of scenarios, 100
documentation of, 277–279
making, 274–276, 277
silent, 275–276
techniques, 275
validation of, 280
- observers, 300
- obstacles
to group work, 284, 285
identification of, 67
workshops, 67–68
- office carpeting, 224–225

- onion model, 28, 29, 37, 79, 294
 open systems standards, 346
 operational qualities, classification of, 140
 operational roles, 38, 48
 operational scenarios, 115, 116, 118–119
 operational stakeholders
 requirements from, 47
 roles of, 199–200
 operational time, 151
 operations, talking through, 276–277
 operator actions, 102–103
 operators, 80, 81
 maintenance, 30
 normal, 30, 48
 support, 30
 optioneering, 344–366
 discussion of options, 352–353
 iteration of, 350
 life cycle, 345–350
 with PCA, 360–366
 problem, criteria and solving, 360–362
 process, 350–352
 requirement elements discovered from, 366
 requirements needed before and after, 347–350
 selection of the winning design, 352–360
 tips for, 365
 organisation
 of product functions, 386–387
 of requirements
 specification, 385–394
 output events, 91–92
 output priorities, 236, 243–254
 discovery of, 243–250
 documentation of, 251–253
 validation, 253–254
 owner/managers, goals for a restaurant, 57
 performance, 67, 212
 of functions, 212–214
 measures of, 154
 targets, 153–154
 of use cases, 215–216
 photographs, in documenting observations, 279
 physical constraints, 145–146
 ‘physical equivalence’, 192
 Pinker, Steven, 16
 Plato, 177
 Polanyi, Michael, 263
 policies, 52
 price, 355–356
 principal components analysis (PCA), 359–360
 biplot of criteria on option chart, 365
 decision-making with, 362–366
 in theory and practice, 364
 principle of commensurate care, 112
 priorities, 235–256, 392
 checking against scenario and infrastructure dependencies, 243
 deciding on top priority, 243
 documentation of, 394
 input, 236, 237–243
 kinds of, 236
 output, 236, 243–254
 in retail IT project case study, 378
 techniques for obtaining, 238
 prioritisation, 198
 Problem Pyramid™, 11–12
 problem reports, requirements from, 333–334
 problems, cause(s) of, 12
 procedures, 32
 process bias, 375
 process qualities, 146–147
 producibility, 146
 product design, 12
 product development
 from analogous products or concepts, 336–337
 on a ‘brown field site’, 331–332
 from competitors’ products, 335–336
 dominant requirements and discovery methods, 377
 from earlier version of a product, 332
 planning, 377
 from problem reports, 333–334
 from simulators, 332–333

- product documentation, requirements
from, 334–335
- product events, 89
- product launch, 221
- product managers, 48, 254
- product qualities, 147–157
- product requirements, 11
- product roadmap, 252, 254
- product variants, documentation of
output priorities, 252, 253
- products, 80, 81
organisation of functions, 386–387
- project dictionary, 190, 191
construction of, 194–195
tips for, 198
validation of, 204–205
- project management,
requirements-driven, 381–385
- project managers, 384–385
- project wall, 305–306, 313
- project website, 306, 307, 313
- project Wiki, 307–308
- projects
finding the right process for, 376–385
role of group media, 312–314
scope of, 76
types of, 9, 10
- prototypes
. *see* requirements prototypes
- prototyping
. *see* requirements prototyping
- public, operational roles, 48
- purchasers, 38, 48
- qualitative criteria, 361
- qualities, 296, 382
acceptance criteria, 216–218
analysis of stakeholders to
discover, 136
definition of, 132
discovery of, 133–141
documentation of, 141–157, 393
that govern choices, 132–133
validation of, 157–159
- quality function deployment (QFD), 356
matrix, 356–358
- quality goals, 58–59, 136
- quality of service (QoS), 223–230
accuracy/correctness, 232
availability, 232
measurements, 225, 228–230, 378
measures, 231–232
speed/performance, 232
tips for choosing measures of, 231
validation of, 230–233
- questionnaires, 260
- railway project, 8–9
- rationale, 161–188, 296, 352–353, 357
discovery of, 163–169
documentation of, 169–183, 394
models, 178–182, 392
validation of, 183–187
value of, 162–163
walkthroughs, 184
- rationalisation, of a set of
requirements, 166–168
- real problems, identification of, 12
- real-time and embedded systems,
dominant requirements elements
and discovery methods, 377
- reasoning, 177
- rebuttals, 185
- recording
with digital cameras, 300–301
workshops, 299–301
- regulations and standards, 32–33
- regulators, 32, 38, 48
- reliability, 67, 148–149
- remote demonstrations, 324–325
- required care, 109
- requirement elements, 383
balanced mix of, 393
and discovery methods, 393–394
network of, 16–18
product development, 377
- requirement managers, triage, 253
- requirements
activities after discovery, 374–375
affect of design options on, 348
from analogous products or
concepts, 336–337
on a ‘brown field site’, 331–332
- business, 11

- definition of, 16
- dependencies of, 247–250
- differences between goals and, 54
- discovery by independent experts, 239
- from earlier version of a product, 332
- from groups, 283–316
- from individuals, 259–282
- iteration of, 346–347
- legal, 239
- life history, 245–246, 247
- from non-operational stakeholders, 47
- from operational stakeholders, 47
- organisation of, 390
- prioritised, 52
- from problem reports, 333–334
- product, 11
- from product documentation, 334–335
- and project management, 381–385
- rationalising a set of, 166–168
- real, 10, 11
- representation in a class model, 203
- reusable sets of, 33
- satisfying a set of, 347
- for a service, 225
- sources of, 10
- state transition diagram, 245
- table-driven, 90–91
- from things, 317–341
- validation of, 340
- verification of, 52, 157
- voting for, 239, 240
- workshops, 284
- requirements analysts, 13
- Requirements by Collaboration* (Gottesdiener), 296
- requirements capture, 7
- requirements cycle, 14
- requirements database tools, 391
- requirements ‘engineering’, 15
- Requirements Engineering* (Kotonya and Sommerville), 13
- Requirements-Led Project Management* (Robertson), 384
- requirements management (RM) tools, 309–310, 313
- requirements prototypes, 318–329
 - demonstration of, 322–329
 - making, 322, 323
 - observation of, 276
 - purpose of, 319
 - screens, 320, 326–327
 - techniques, 319–329
 - tips for, 329
- requirements prototyping, 201, 214, 263, 318–329
 - cycle phases, 320, 321
 - stakeholders wants and, 320–322
- requirements reuse, 318, 337–339
 - naïve reuse, 337–338
 - product lines, 338
 - standardisation, 338
 - tool support, 338–339
- requirements specification, 385–394
 - levels of, 385–386
 - relating different types of, 388–390
 - templates, 385
 - traditional ‘shall’s’, 387–388
 - use cases, 386, 387
- requirements tools, 390–391
- resources, competing for, 250
- responsibilities, roles and, 199–200
- restaurant, goals for a, 57–59
- retail business software, design options affect on requirements, 348
- retail IT project case study, 377–379
 - elements/contexts matrix, 380
 - information model, 379
- retail system, roles in, 199, 200
- reuse, of requirements, 337–339
- reverse engineering, 318, 330–337
- review comments, 334
- rich pictures, 77–78, 80, 295
- risks, 168–169, 172, 355
- Robertson, Suzanne and James, 384
- role/action lists, 101–102
- roles
 - operational stakeholders, 199–200
 - and responsibilities, 199–200
 - in a retail system, 199
- rule book, 265

- safety, 66, 140, 152, 182
safety-related systems, 110
samples, input prioritisation and, 241
satisfaction argument, 175–176
scenarios, 97–129, 263, 382, 389
 acceptance criteria, 214–215
 acting out scenes, 107
 analysis, 392
 calling up, 91
 checking, 127
 continuous load (soak test), 112
 data handling, 106
 day in the life of (DILO), 105, 112
 defining, 103
 dependencies, 247–248
 discovery of, 98–114
 documentation of, 114–123, 393
 exception-handling, 110
 failure injection, 112
 illustrating, 103
 negative, 107–114
 operational, 118–119
 peak load (stress test), 112
 playing through, 126
 standard patterns, 105–106
 transaction/operation/task, 106
unwanted, 113–114
validation of, 124–127
walkthroughs, 124–126
whole life, 105
workshops, 101–107, 296
- Schrage, Michael, 328
- scientific software, dominant
 requirements and discovery
 methods, 377
- scope
 documentation of, 393
 as a list of events, 87–89
 creep, 76
- scree plots, 363, 364
- scribes, 300
- security, 152
 measuring, 217–218
- security threats, 33, 67
- Serious Play* (Schrage), 328
- services, 81
 contracts, 226
- requirements for carpeting, 225
‘show and tell’ events, 304
silent observation, 275–276
Simon, Herbert, 347
simulation, 214
simulators, product development
 from, 332–333
- singular value decomposition (SVD), 363
- social benefits, 355
- soft goals, 59
- soft processes, 12–16
- soft systems, 79
- soft systems methodology, 13–14
 for ill-defined boundaries, 77–87
Soft Systems Methodology (Checkland and Scholes), 77
- software
 demonstrations of, 325–326
 prototyping and, 329
 table-driven, 90–91
Software Cinema (Bruegge), 279
- software dependability, 147
- software functions, 66
- software problem reports, 334
- software products, sorting out priorities
 from reports of, 240
- software projects
 acceptance tests, 213
 checklists, 139–140
 opposition risk, 34
 subset of NFRs, 139
- spacecraft, goals for, 54–58
- specifications, 393
- sponsors, 34–35
- stakeholders, 13, 27–49
 analysis, 382, 391
 discovery of, 28–36, 41
 and discovery of qualities and
 constraints, 136
- documentation of requirement
 elements, 393
- engaging with, 41
- goals, 237
 goals for a tram service, 62
 hostile roles, 38
 hybrid roles, 38

- identification of, 37–41
influences on product development, 42–43
interfacing roles, 38, **48**
interviews with, 261–274
involvement in projects, 45
keeping track of, 42
maintenance roles, **48**
management of, 41–45
manufacturing roles, **48**
marketing roles, **48**
needs of, 249
negative, 33–34, 38
onion model, 28, 29, 37, 79, 294
operational roles, 38, **48**
prioritising, 43–44
requirements, 14
restaurant goals, 57–59
in retail IT project case study, 378
roles, 38–39, **48**
in a tram service, 35–36
validation of, 45–46
standards, 32–33, 201
'guidance', 32, 33
open systems, 346
reuse of, **392**
using to identify qualities, 138–139
statute laws, 32
Stevens, Richard, 15, 17
stock controller, 199, **200**
stories, 114, **116**
storyboards, 114, 116–118
illustrating a negative scenario, 117
tips for, 118
and user interface design, 119
uses of, 117
storytelling, for discovery of scenarios, 99–101
strategic dependency diagrams, 59
strategic rationale diagrams, 60
- The Stuff of Thought* (Pinker), 16
success guarantees, 215
suggestion forms, 334
suppliers, 58
support roles, **48**
surrogate roles, 38, 39–40, **48**
- surveys, 241
survivability, 153
sustainability, 144
swimlanes, 103–105
for a highly automated restaurant, **104**
in real time, 105
synonyms, 191–193
system boundaries, 80
system dependability, 147
system requirements, 15
systems, 80, **81**
hard and soft, 79
operational stakeholders, 30
systems engineers, 13
tables, representation of requirements in, 90–91
tacit knowledge, **263**, 282, 322
targets, prediction of meeting, 158
telecommunications utility, exception scenarios, 111
templates, 385, 389, **392**, 423–427
testability, 147, 222–223
things
requirements from, 317–341
validation of requirements from, 340
threats
discovery of, 112
identification of, 67
workshops, 67–68
time-triggered events, 88, 89
tools
modelling, 308–309
requirements management (RM), 309–310, **313**
for working at a distance, 310–312
Toulmin, Stephen, 177
toys, using during workshops, 294
traceability, 162, 173–177, 358
analysis of, 184–186
to parent requirements, 174
traceability tools, 390–391
trade-offs, 84, 168, 219, 298, 343–371, **392**
commercial off-the-shelf (COTS) products, 350
engineering of, 344–366

- trade-offs, (*continued*)
prioritisation, 382
requirements-first life-cycle
 myth, 344–345
in retail IT project case study, 378
validation of, 367
- traditional context diagrams, 87, 88
- traditional ‘shall’s’, 387–388
- train control box, testing of, 8–9
- tram service
 goals and trade-offs, 59–62
 side effects of, 61
 stakeholders, 35–36
- transaction/operation/task, 106
- transport planning case study, 379–381
 elements/contexts matrix, 381
 issues, 380
- triage, 237–238, 251
 on output priorities, 244–245
- trust
 building and sustaining, 302–304
 sustaining after the workshop, 304
- type tests, 148
- UML constraints, 202
- UML Distilled* (Fowler), 271
- Unix operating system, 248
- unwanted scenarios, 113–114
- usability, 154–156
 acceptance criteria, 218
- usability panels, 155
- usability testing, 155
- usage qualities, 147–157
 availability and reliability, 148–149
 dependability, 147–148
 interoperability, 156–157
 maintainability, 149
 performance targets, 153–154
 programme requirements, 157
 safety, 152
 security, 152
 survivability, 153
 usability, 154–156
- USB (Universal Serial Bus), 84
- use case diagrams, 92
- use cases, 98, 115, 116, 119–123
 acceptance criteria, 214–215
- approximate effort to prepare, 122
- documentation of, 122
- ‘fully dressed’ use case template, 121
- inside-out, 123
- organisation of, 122–123
- performance of, 215–216
- requirements specification, 386, 387
- in retail IT project case study, 378
- structure and behaviour, 120
- validation of, 125
. *see also* misuse cases
- user interface design, storyboards
 and, 119
- user interface guidelines, 155
- user interface prototypes, 324
- user stories, 115
- User Stories Applied* (Cohn), 20
- The Uses of Argument* (Toulmin), 177
- validation
 of acceptance criteria, 222–223
 of boundary choice, 86–87
 of data models, 205
 of input priorities, 242–243
 of interfaces and events, 93–94
 of interview findings, 273–274
 of observations, 280
 of output priorities, 253–254
 of project dictionary, 204–205
 of qualities and constraints, 157–159
 of quality of service (QoS), 230–233
 of rationale and
 assumptions, 183–187
 of requirements from things, 340
 of scenarios, 124–127
 of trade-offs, 367
 of use cases, 125
 of workshop findings, 302–305
- value engineering, 355
- verification methods, 219–221
 analysis, 221
 certificate of testing, 220
 demonstration, 220–221
 simulation, 221
 tests, 220
- verification, of requirements, 157
- video conferencing, 312

-
- video, use in documenting observations, 279
 - vision statements, use of word ‘will’ in, 165–166
 - vision, 53
 - walkthroughs, 242–243
 - warrants, 185
 - weighting design options, 353–360
 - analytic hierarchical process (AHP), 356
 - challenges to, 354–355
 - naïve weighting, 354
 - quality function deployment (QFD), 356
 - matrix, 356–8
 - ‘when’ requirements, 89–90
 - whole life scenarios, 105
 - Wiki, 307–308, 313
 - Word, 171
 - workshops, 63, 64, 263, 285, 286–305
 - activities, 64–65, 294–297
 - building and sustaining trust in, 303–304
 - combination, 297
 - complex topics, 297–298
 - for discovery of scenarios, 100, 101–107
 - engaging participants in, 293–294
 - ‘house rules’, 291
 - identification of negative issues, 295
 - large groups, 298
 - mindset and skills for requirements workshop, 288–289
 - mission of, 286
 - negative scenarios (for QoS), 232
 - obstacles and threats, 67–68
 - outputs, 295
 - planning, 287–289
 - pros and cons of, 313
 - recording, 299–301
 - rehearsal, 289–290
 - resources, 290
 - roles, 291
 - room setup, 290–291
 - setup of, 290–299
 - solutions to problems in workshop rooms, 292
 - sustaining trust after, 304
 - as theatre, 287
 - tips for, 297
 - validation of findings, 302–305
 - Writing Better Requirements* (Alexander and Stevens), 17
 - Writing Effective Use Cases* (Cockburn), 120
 - Young, Ralph, 345, 346–347, 382, 384–385